Thèse
présentée pour obtenir le grade de
Docteur de l'École polytechnique

Spécialité :
Mathématiques – Informatique

par
Romain Lebreton

Contributions à l'algorithmique détendue et à la résolution des systèmes polynomiaux

Soutenue le 11 décembre 2012 devant le jury composé de :

M. Richard Brent	Australian National University	Rapporteur
M. Marc Giusti	CNRS & École polytechnique	Directeur
M. Bernard Mourrain	Inria Sophia Antipolis - Méditerranée	Rapporteur
M. Jean-Michel Muller	École Normale Supérieure de Lyon	Examinateur
M. Éric Schost	University of Western Ontario	Directeur
M^{me} Annick Valibouze	Université Paris 6	Examinateur
M. Kazuhiro Yokoyama	University of Rikkyo	Rapporteur
M. Paul Zimmermann	Inria Nancy - Grand Est	Examinateur

Remerciements

Ma première pensée va naturellement à mes deux directeurs de thèse Marc Giusti et Éric Schost. Marc, merci pour tes leçons patientes de géométrie, pour ton soutien tout au long de ces trois années et pour ton sens de l'humour pince-sans-rire. Faire une thèse, c'est apprendre l'indépendance et je te sais gré de la liberté que tu m'as laissée, à l'image de ta propre expérience de la thèse.

Les voyages forment la jeunesse, et mes nombreux séjours à London auprès d'Éric furent à chaque occasion des catalyseurs de mes recherches. Éric, il faut dire que je me retrouve dans ta façon d'appréhender la science, de toujours naviguer entre pensée mathématique et expérimentation informatique sur un exemple « concret ». Alors merci pour avoir su toujours trouver le temps, le bon éclairage sur nos questions, ainsi que les mots justes pour remotiver les troupes.

Je suis reconnaissant envers l'École polytechnique pour m'avoir hébergé au laboratoire d'informatique LIX, m'avoir rémunéré *via* une bourse internationale Gaspard Monge et également pour m'avoir permis de faire mon monitorat sur place. Cette thèse a aussi en partie été soutenue par le projet MaGiX ANR-09-JCJC-0098-01 et l'allocation Digiteo 2009-36HD. Merci aussi à l'University of Western Ontario pour son accueil et son financement de mes visites.

Je remercie vivement Richard Brent, Bernard Mourrain et Kazuhiro Yokoyama d'avoir accepté d'être rapporteurs de cette thèse. Merci à Jean-Michel Muller, Annick Valibouze et Paul Zimmermann pour leur présence dans le jury. Merci aussi aux rapporteurs anonymes des articles écrits lors de cette thèse qui ont contribué à l'amélioration de mes travaux.

Par ordre chronologique, je tiens à remercier les gens qui ont particulièrement contribué à ce que je trouve ma voie. Je commence par Nicolas Tosel dont les excellents cours de prépa et la dévotion à l'enseignement m'ont grandement aidé à intégrer l'É.N.S. Ulm. Parce qu'il a été le premier à m'enseigner le Calcul Formel lors de ma préparation à l'agrégation, je tiens à remercier Pierre-Vincent Koseleff. Une discipline qui mêle algèbre et informatique, je crois que je serais tombé dedans tôt ou tard. Une partie de la révélation est venue d'un court entretien avec mon tuteur en troisième année d'É.N.S. : Wendelin Werner. Lorsqu'il m'a expliqué que son attrait pour les probabilités datait de sa jeunesse, il m'est apparu évident que je ne pourrais faire de mathématiques sans informatique. L'année qui suivit, fut marquée par Alin Bostan, Bruno Salvy et Frédéric Chyzak dont la passion communicative pour le Calcul Formel allait de pair avec leur remarquable cours. C'est ainsi que j'ai commencé ma thèse, voulant mieux comprendre la géométrie algébrique qui m'avait tant intrigué à l'É.N.S., mais grâce au point de vue toujours pragmatique du Calcul Formel.

Au sein du laboratoire d'informatique LIX, je me dois de commencer par Jérémy BERTHOMIEU avec qui j'ai passé le plus clair de mes trois années de thèse. Tu as largement contribué à faire de notre bureau un lieu convivial. Avec ton âme de grand sportif, mais non-pratiquant, tu resteras à mes yeux un FEDERER de l'orthographe et du LaTeX. Merci à Grégoire LECERF pour nos discussions scientifiques et à Joris VAN DER HOEVEN pour ton fourmillement d'idées dépassant allègrement le cadre du Calcul Formel (TeXmacs, utilisé pour écrire cette thèse, est un exemple parmi tant d'autres). Une pensée à François OLLIVIER et à Jean MOULIN-OLLAGNIER qui ont accueilli deux squats dans leur bureau, et plus généralement à tous les gens que j'ai rencontrés lors d'un passage, plus ou moins long, au LIX et avec qui j'ai échangé avec plaisir : Antoine COLIN, Philippe TRÉBUCHET, Daouda N. DIATTA, Denis RAUX, François POULAIN, Miguel DE BENITO, Pierre LAIREZ et Olivier SCHWANDER. Merci à mes collègues d'enseignement, notamment Philippe CHASSIGNET, Jean-Christophe FILLIÂTRE et Stéphane LENGRAND.

Une pensée pour les autres ~~galériens~~ thésards de Calcul Formel Luca DE FEO, Jean-François BIASSE, Marc MEZZAROBBA, Guillaume « COINTIN », Alexandre BENOIT, Morgan BARBIER, Guillaume MOROZ, Adrien POTEAUX : heureusement qu'on n'est pas seul à ramer ! Un grand merci aux assistantes Corinne POULAIN, Sylvie JABINET et Évelyne RAYSSAC ainsi qu'à James REGIS et Elie MABO du service informatique pour le temps passé à m'aider. Je remercie aussi vivement les Montpelliérains Laurent IMBERT, Pascal GIORGI et Eleonora GUERRINI qui m'ont immédiatement intégré et soutenu pour ma fin de thèse.

Je n'oublie pas les gens de London : Javad DOLISKANI, Marc MORENO MAZA, Muhammad CHOWDHURY, Esmaeil MEHRABI, Livio ZERBINI et Vadim MAZALOV. Et une pensée particulière pour Ephie TSIAPALIS qui a été comme une mère pour moi pendant les hivers canadiens.

À tous mes super-potes du Groupe de Travail « viens boire un p'tit coup à la maison » de l'É.N.S., spéciale dédicace à Mathieu « shorty » HURUGUEN, Oolivier BENOIST, Thomouise BENOÎT-HAETTEL, Morgane (et Romain) CARIBOU et son inénarrable P.-l.-P. DUBLANCHET sans oublier Pierre, Simon, Loïc, Michal, Nicolas, Adeline, Damien, Yoann et David...

« Je sers la science et c'est ma joie » dirait le disciple dans Léonard, mais heureusement pas la seule joie. Je remercie tous mes partenaires de football pour les bons moments passés ensemble.

Un grand merci à ma famille qu'elle soit de Saint-Raphaël, de Lyon, de Brest, de Villejuif, de Gauré ou d'ailleurs, pour leur soutien inconditionnel et leur tolérance à l'Alzheimer des anniversaires dont je semble souffrir.

Et la meilleure pour la fin : merci Isabelle. À deux, c'est toujours mieux !

Table des matières

Introduction

An English version of this introduction can be found in Appendix.

De nos jours, la puissance croissante des moyens de calcul est mise à profit, entre autres, pour améliorer la précision de solutions existantes à certains problèmes ou pour traiter des défis avec des entrées plus conséquentes. Cependant, il est fréquent en Informatique que le temps de calcul d'un algorithme augmente bien plus rapidement que la précision voulue pour les solutions. Une stratégie efficace pour contrer cet effet négatif consiste à découper un gros problème en de multiples petits problèmes, de les résoudre puis reconstruire la solution du problème initial.

En Calcul Formel, le théorème des restes chinois permet un tel découpage. Divisons un problème en plusieurs problèmes similaires modulo des entiers n_1, ..., n_r premiers entre eux. Alors le théorème des restes chinois permet de retrouver la solution du problème initial à partir des solutions des bouts de problèmes. Ce schéma de calcul, quand il fait sens, permet de calculer en temps proportionnel à la précision plus une reconstruction multi-modulaire.

Les techniques multi-modulaires peuvent être utilisées de concert avec une remontée p-adique. Prenons l'un de ces éléments premiers entre eux, par exemple n_1, de la forme $n_1 = p^\ell$. Réduisons la taille du problème en ne le résolvant que modulo p. On appelle *remontée p-adique* la reconstruction de la solution modulo p^ℓ à partir de celle modulo p. Ainsi, la résolution d'un problème modulo p^ℓ se réduit à la résolution d'un petit problème et à une remontée p-adique. Dans la mesure où la remontée peut être plus facile à calculer que la résolution directe du problème complet, l'utilisation d'une remontée p-adique peut réduire considérablement le temps de calcul.

Cette thèse est en majeure partie dédiée au calcul rapide de remontée p-adique par un type d'algorithmes récent, les algorithmes détendus.

* * *

Mon travail se situe dans la lignée de la série d'articles sur les algorithmes détendus initiés par van der Hoeven [Hoe97, Hoe02, Hoe03, Hoe07, Hoe09] pour les séries formelles et adaptés par Berthomieu, Lecerf et van der Hoeven [BHL11] au cas des anneaux p-adiques généraux. Un p-adique a est une suite infinie de coefficients $(a_i)_{i\in\mathbb{N}}$ que l'on écrit $a := \sum_{i\in\mathbb{N}} a_i\, p^i$.

Les algorithmes détendus sont un cas particulier d'algorithmes en-ligne. Les algorithmes en-ligne furent créés par Hennie [Hen66] dans le modèle de Turing. Un *algorithme en-ligne* qui prend en entrée des p-adiques est un algorithme qui lit les coefficients du p-adique un à un, et produit le n-ième coefficient de la sortie avant de lire le $(n+1)$-ième coefficient de l'entrée.

Par exemple, la multiplication de deux p-adiques par un algorithme en-ligne prend, à première vue, un temps quadratique en la précision. Cependant il existe un algorithme de multiplication en-ligne quasi-optimal que l'on peut trouver dans [FS74, Sch97, Hoe97].

Le principal avantage des algorithmes en-ligne est qu'ils permettent la remontée de p-adiques *récursifs*. Un p-adique récursif y est un p-adique qui vérifie $y = \Phi(y)$ où Φ est un opérateur tel que le n-ième coefficient du p-adique $\Phi(y)$ ne dépend pas des coefficients d'indice supérieur ou égal à n. Par conséquent, y peut être calculé récursivement à partir de la donnée de y_0 et de Φ.

Dans les articles [Wat89, Hoe02] se trouve un algorithme qui calcule y à partir de son équation de point fixe $y = \Phi(y)$. Un aspect fondamental de cet algorithme est que *son coût est celui de l'évaluation de* Φ *par un algorithme détendu*. En utilisant l'algorithme de multiplication en-ligne rapide, l'article [Hoe97] obtient un cadre de travail efficace pour calculer avec les p-adiques récursifs. Comme van der Hoeven n'était apparemment pas au courant des travaux antérieurs sur les algorithmes en-ligne, il nomma son algorithme « multiplication détendue » et les algorithmes suivants furent appelés algorithmes détendus. Ainsi, on peut rétrospectivement définir un algorithme détendu comme un algorithme en-ligne rapide. Les algorithmes détendus sont aussi liés aux algorithmes paresseux que l'on peut voir comme des algorithmes en-ligne qui minimisent le coût de chaque étape. En résumé, les algorithmes détendus et paresseux sont deux cas d'algorithmes en-ligne, le premier minimisant le coût globalement et le second localement. À partir de maintenant, nous utiliserons ces définitions et leurs adaptations aux anneaux p-adiques généraux [BHL11].

Une contribution principale de cette thèse est l'utilisation d'algorithmes détendus dans le contexte de la résolution d'équations par remontée p-adique. Ce contexte s'applique, entre autres, aux types de systèmes standards et importants suivants : linéaire, algébrique et différentiel.

Les méthodes en-ligne pour la remontée de p-adiques récursifs doivent être comparées à l'autre grande famille d'algorithmes de résolution d'équations par remontée p-adique. Quand un p-adique y est donné par une équation implicite $f(y) = 0$ dont la dérivée en y est inversible, l'opérateur de Newton-Hensel s'applique et calcule y à toutes précisions à partir de son premier coefficient p-adique y_0. Cet opérateur a été introduit par Newton dans [New36] et a été adapté dans le contexte des entiers p-adiques par Hensel [Hen18]. L'opérateur de Newton ne s'applique pas directement à toutes les situations. Cependant, le principe sous-jacent peut souvent être adapté (*cf.* Chapitres 4 et 6).

Comme nous le remarquerons en de nombreuses occasions dans ce mémoire, la remontée par des algorithmes en-ligne nécessite, asymptotiquement en la précision, moins de produits en-ligne que les algorithmes hors-ligne ne nécessitent de multiplications hors-ligne. *A contrario*, une multiplication en-ligne est plus coûteuse qu'une multiplication hors-ligne. Nous implémenterons donc la plupart de nos algorithmes pour comparer les temps en pratique.

* * *

Cette thèse explore la remontée p-adique en-ligne de solutions de différents types de systèmes d'équations. La partie I présente les algorithmes détendus et leur application aux p-adiques récursifs. Le chapitre 1, après avoir donné la définition des algorithmes en-ligne, rappelle plusieurs algorithmes en-ligne rapides de multiplication de p-adiques et en présente un nouveau. Ensuite, le chapitre 2 met en place le cadre de travail dans lequel les p-adiques récursifs sont calculés par des algorithmes en-ligne. Nous utiliserons ce cadre dans les parties II et III pour donner de nouveaux algorithmes en-ligne pour la remontée p-adique de solutions de différents types de systèmes d'équations.

La partie II est centrée sur les systèmes linéaires. Au cours du chapitre 3, nous débutons par l'algèbre linéaire sur les p-adiques puis poursuivons par la résolution de systèmes linéaires en prenant en compte le type de représentation des matrices. Puis nous fournissons des algorithmes en-ligne pour calculer les séries formelles solutions d'une large classe d'équations différentielles ou (q)-différentielles dans le chapitre 4.

La partie III est dédiée à la remontée p-adique de solutions de systèmes algébriques, par ordre croissant de généralité. Le chapitre 5 donne un algorithme en-ligne pour la remontée p-adique de racines régulières de systèmes algébriques. Le chapitre 6 traite quant à lui de la remontée de représentations à une variable et d'ensembles triangulaires par des algorithmes en-ligne.

Enfin, la partie IV traite un cas particulier des systèmes algébriques, celui de l'idéal des relations symétriques. Pour ce problème, avant de commencer la remontée, le calcul d'une représentation à une variable à précision 0 pose déjà problème. Nous donnons dans le chapitre 7 un algorithme quasi-optimal pour calculer une telle représentation à une variable et nous utilisons cette représentation pour calculer efficacement dans l'algèbre quotient correspondante.

Dans l'appendice A, nous prouvons que la remontée des communément appelés « invariants fondamentaux » modulo p vers les rationnels est triviale.

Contributions En résumé, les contributions originales de cette thèse sont :

1. un nouvel algorithme détendu de multiplication de p-adiques ;
2. une analyse de complexité précise de plusieurs algorithmes détendus de multiplication ;
3. un cadre exact pour calculer des p-adiques récursifs par des algorithmes en-ligne ;
4. un solveur détendu de systèmes linéaires p-adiques ;
5. deux algorithmes de remontée de séries formelles solutions d'équations singulières aux (q)-différences : l'un est détendu, l'autre est hors-ligne et adapte l'opérateur de Newton ;
6. un algorithme détendu pour la remontée de solutions p-adiques régulières de systèmes algébriques ;
7. plus généralement, un algorithme détendu pour la remontée d'ensembles triangulaires et de représentations à une variable ;

8. une présentation des premiers algorithmes quasi-linéaires qui calculent une représentation à une variable de l'algèbre de décomposition universelle sur les corps finis d'une part et le polynôme caractéristique de ses éléments d'autre part ;
9. enfin, une preuve qu'il suffit, pour calculer des invariants fondamentaux sur les rationnels, de les calculer modulo p.

Publications Les contributions 3, 4 et 6 ont été publiées dans les actes de la conférence *ISSAC'12* avec J. Berthomieu [BL12]. Leur présentation dans ce mémoire contient des détails, preuves et exemples additionnels. De plus, l'application du solveur détendu de systèmes linéaires du point 4 aux matrices structurées est un travail en cours, fait en commun avec É. Schost. La contribution du point 5 provient d'un article écrit avec A. Bostan, M. Chowdhurry, B. Salvy et É. Schost qui est publié dans les actes de *ISSAC'12* [BCL+12]. Dernièrement, les contributions de l'item 8 viennent d'un travail avec É. Schost et publié dans les actes de *ISSAC'12* [LS12].

Prix Le prix du meilleur article étudiant m'a été décerné à la conférence *ISSAC'12* pour l'article [LS12]. J'ai également reçu à cette conférence le prix du meilleur poster décerné par le « Fachgruppe Computer Algebra » pour le poster [LMS12] en collaboration avec E. Mehrabi et É. Schost.

1 Algorithmes détendus pour la multiplication

Cette section présente la notion d'algorithmes en-ligne et détendus sur des anneaux p-adiques généraux. Soit R un anneau commutatif avec unité. Étant donné un idéal principal propre (p) avec $p \in R$, nous notons R_p la complétion de l'anneau R pour la valuation p-adique. Les éléments $a \in R_p$ sont appelés des p-adiques. Pour avoir l'unicité de la décomposition des éléments de R_p, fixons un sous-ensemble M de R tel que la projection $\pi\colon M \to R/(p)$ soit une bijection. Alors, tout p-adique $a \in R_p$ s'écrit de manière unique $a = \sum_{i\in\mathbb{N}} a_i p^i$ avec $a_i \in M$.

Deux exemples classiques sont l'anneau des séries formelles $\Bbbk[[X]]$ qui est le complété de l'anneau des polynômes $\Bbbk[X]$ pour l'idéal (X), et l'anneau des entiers p-adiques $\mathbb{Z}_p$ qui est le complété de l'anneau des entiers $\mathbb{Z}$ pour l'idéal (p). Dans le cas, $R = \mathbb{Z}$, nous prendrons $M = \{-(p-1)/2, \ldots, (p-1)/2\}$ si $p \neq 2$ et $M = \{0, 1\}$ si $p = 2$. Pour $R = \Bbbk[X]$, nous prendrons $M = \Bbbk$.

Nous sommes maintenant en mesure de donner la définition d'un algorithme détendu telle qu'elle fut introduite par [Hen66].

Définition 1. ([Hen66, FS74]) *Soit une machine de Turing qui calcule une fonction f sur des suites, avec $f\colon \Sigma^* \times \Sigma^* \to \Delta^*$ et Σ et Δ des ensembles. On dit que la machine calcule f* en-ligne *si, pour toutes suites $a = a_0 a_1 \ldots a_n$, $b = b_0 b_1 \ldots b_n$ en entrée et pour des sorties correspondantes $f(a, b) = c_0 c_1 \ldots c_n$, avec $a_i, b_j \in \Sigma$, $c_k \in \Delta$, la machine écrit la sortie c_k avant de lire soit a_j, soit b_j, avec $0 \leqslant k < j \leqslant n$.*

La machine calcule f semi-en-ligne *(par rapport au premier argument) si elle écrit la sortie c_k avant de lire a_j pour $0 \leqslant k < j \leqslant n$. L'entrée a est appelée l'entrée* en-ligne *et b l'entrée* hors-ligne.

Le reste de ce chapitre traite des algorithmes en-ligne rapides pour la multiplication de p-adiques. Ces algorithmes sont faits d'appels à des multiplications hors-ligne de p-adiques de précision finie. Nous rappelons d'abord l'état de l'art des algorithmes de multiplication de polynômes et d'entiers qui sont respectivement les séries formelles et les entiers p-adiques de précision finie. Nous regarderons aussi les algorithmes existants pour les produits court et médian.

Avec ces notions à portée de main, nous faisons un rappel des algorithmes détendus suivants de multiplication de p-adiques. Le premier algorithme en-ligne quasi-optimal de multiplication fut présenté dans [FS74] pour les entiers, puis dans [Sch97] pour les nombres réels et finalement dans [Hoe97] pour les séries formelles. Ce dernier algorithme fut adapté en un algorithme semi-détendu (ou semi-en-ligne) dans [FS74, Hoe03]. Un première amélioration d'un facteur constant du produit semi-détendu est présenté dans [Hoe03].

Notre contribution consiste en la présentation d'un nouvel algorithme détendu utilisant des produits *médians* et *courts* qui améliore d'un facteur constant les algorithmes précédents. De plus, nous analysons pour la première fois précisément le nombre de multiplications de base que font ces algorithmes détendus. Pour finir, nous donnons des temps de calcul qui confirment le bon comportement des algorithmes détendus qui utilisent le produit médian.

À partir de maintenant, nous utiliserons les notations suivantes. Pour tout p-adique $a = \sum_{i\in\mathbb{N}} a_i\, p^i$, la longueur $\lambda(a)$ de a est définie par $\lambda(a) := 1 + \sup(i \in \mathbb{N} \mid a_i \neq 0)$ si $a \neq 0$ et $\lambda(0) = 0$. Le coût de la multiplication de deux p-adiques de longueur N par un algorithme hors-ligne (resp. en-ligne) est noté $\mathsf{I}(N)$ (resp. $\mathsf{R}(N)$) dans notre modèle de complexité précisé en Section 1.1.2. Aussi nous noterons $\mathsf{M}(N)$ le nombre d'opérations arithmétiques que nécessite la multiplication de deux polynômes de longueur N.

2 Nombres p-adiques récursifs

L'étude des algorithmes en-ligne est motivée par l'implémentation pratique des p-adiques récursifs à l'aide de ces algorithmes. Il fut pointé pour la première fois dans [Wat89] que le calcul de séries formelles récursives est bien adapté à l'algorithmique paresseuse. Cela donna lieu à l'époque à un cadre très pratique pour calculer avec les séries formelles récursives, mais pas encore très efficace.

Ce fait fut redécouvert par [Hoe02] plus généralement pour l'algorithmique en-ligne. Mis bout à bout avec l'algorithme en-ligne rapide de multiplication de [Hoe97], van der Hoeven obtint un cadre simple et efficace pour calculer avec les séries formelles récursives.

Un p-adique récursif y est un p-adique qui satisfait $y = \Phi(y)$ pour un opérateur Φ qui vérifie l'égalité entre les n-ièmes coefficients p-adiques $\Phi(y)_n = \Phi(y + a\, p^n)_n$ pour tout $a \in R_p$. Par conséquent, y est uniquement déterminé par la donnée de Φ et de son premier coefficient y_0.

Dans ce chapitre, nous rappelons la méthode de [Hoe02] qui, à partir d'un algorithme en-ligne qui évalue la fonction Φ, calcule les coefficients du p-adique récursif y l'un après l'autre. Cependant cette méthode ne marche pas toujours tel quel.

Notre contribution est d'identifier le problème sous-jacent et de donner une condition suffisante pour que la méthode précédente marche. Nous n'avons pas connaissance de traces de ce problème dans la littérature.

Nous introduisons la nouvelle notion *d'algorithmes décalés* dans l'optique de résoudre ce problème. Un entier, appelé le décalage, est associé à toute entrée p-adique d'un algorithme donné et mesure quels coefficients de ces entrées sont lus au moment où l'algorithme produit le n-ième coefficient de la sortie. À titre d'exemple, un algorithme est en-ligne si et seulement si son décalage est positif.

Finalement un algorithme est un *algorithme décalé* si son décalage est positif. Nous avons alors à notre disposition les outils nécessaires pour énoncer la proposition fondamentale suivante.

Proposition. *Soient y un p-adique récursif et Ψ un algorithme décalé tels que $y = \Psi(y)$. Alors le p-adique y peut être calculé à précision N dans le temps nécessaire pour évaluer Ψ en y à précision N.*

Ainsi, le coût du calcul d'un p-adique récursif est le même que le coût de sa vérification. La proposition précédente est la pierre angulaire des futures estimations de complexité liées à des p-adiques récursifs. Elle sera utilisée dans les chapitres 3, 4, 5 et 6.

3 Algèbre linéaire sur les p-adiques

Dans ce chapitre, nous présentons un algorithme basé sur le cadre détendu pour les p-adiques récursifs du chapitre 2 qui peut en principe être appliqué à n'importe quel choix de représentation de matrices (dense, creuse, structurée, *etc.*). Nous nous concentrons sur les deux cas particuliers importants que sont les matrices *denses* et *structurées* et nous montrons comment notre algorithme peut améliorer les techniques existantes.

Considérons un système linéaire de la forme $A = B \cdot C$, avec A et B connues et C inconnue. La matrice A appartient à $\mathcal{M}_{r\times s}(R_p)$ et la matrice $B \in \mathcal{M}_{r\times r}(R_p)$ est inversible ; nous résolvons le système linéaire $A = B \cdot C$ en $C \in \mathcal{M}_{r\times s}(R_p)$. Les cas les plus intéressants sont le cas $s = 1$, qui revient à résoudre un système linéaire, et $s = r$, qui contient le problème de l'inversion de B. Une application importante de la résolution linéaire de systèmes à coefficients p-adiques est la résolution de systèmes sur R (c'est-à-dire sur les entiers ou les polynômes par exemple) à l'aide de techniques de remontée.

Nous noterons $d := \max(\lambda(A), \lambda(B))$ la longueur maximale des coefficients des matrices A et B. Soit N la précision à laquelle nous voulons connaître C. Ainsi, nous pourrons toujours supposer que $d \leqslant N$. Le cas particulier $N = d$ correspond à la résolution de systèmes linéaires p-adiques en propre, tandis que la résolution de systèmes linéaires sur R nécessite souvent une précision $N \gg d$. En effet, dans ce cas et pour les séries formelles par exemple, les formules de Cramer indiquent que les numérateurs et les dénominateurs de C ont une longueur $\mathcal{O}(r\,d)$, de telle sorte que l'on doive prendre N de l'ordre de $\mathcal{O}(r\ d)$ pour permettre la reconstruction rationnelle.

Parmi les méthodes préexistantes, un premier algorithme dû à Dixon [Dix82] calcule un à un les coefficients p-adiques de la solution C puis met à jour la matrice A. De l'autre côté de l'étendue des techniques, on trouve l'itération de Newton qui double la précision de la solution à chaque étape. L'algorithme de Moenck-Carter [MC79] est une variante de l'algorithme de Dixon qui travaille avec des p^d-adiques au lieu de p-adiques. Finalement, l'algorithme de Storjohann de remontée à grande précision [Sto03] peut être vu comme une version rapide de l'algorithme de Moenck-Carter, adapté aux cas $N \gg d$.

Nous contribuons avec un algorithme qui résout des systèmes linéaires par des techniques détendues. Cette algorithme est obtenu en prouvant que les coefficients de la solution $C = B^{-1} \cdot A$ sont des p-adiques récursifs. En d'autres termes, nous montrons que C est le point fixe d'un certain opérateur décalé.

Si l'on prend par exemple $s = 1$, le calcul de C à précision N coûte avec notre algorithme à peu près :

1. une inversion $\Gamma := B^{-1}$ modulo (p) ;
2. $\mathcal{O}(N)$ produits matrice-vecteur utilisant l'inverse Γ où chacune des entrées du vecteur est aussi de longueur 1 ;
3. $\mathcal{O}(1)$ produits matrice-vecteur utilisant B, avec un vecteur dont les entrées sont des p-adiques détendus.

Nous verrons que notre algorithme est un intermédiaire entre les algorithmes de Dixon et de Moenck-Carter puisque que nous faisons la même peu coûteuse initialisation modulo (p) que Dixon (*cf.* item 1) tout en gardant la même bonne complexité asymptotique que Moenck-Carter (*cf.* item 3).

Le tableau suivant donne les complexités des algorithmes présentés pour le cas des matrices denses, avec $R_p = \Bbbk[[X]]$, $s = 1$ et les deux valeurs de N importantes en pratique, c'est-à-dire $N = d$ et $N = r\,d$. Il en ressort que pour la résolution à précision $N = d$, notre algorithme est le plus rapide de ceux présentés. En précision $N = r\,d$, la remontée à grande précision de Storjohann est meilleure (car elle est conçue pour de telles précisions). Soit $\omega \in \mathbb{R}_{>0}$ tel que l'on peut multiplier et inverser toute matrice $r \times r$ en $\mathcal{O}(r^\omega)$ opérations arithmétiques sur tout corps.

Algorithme	$N = d$	$N = r\,d$
Dixon	$\tilde{\mathcal{O}}(r^\omega + r^2\,d^2)$	$\tilde{\mathcal{O}}(r^3\,d^2)$
Moenck-Carter	$\tilde{\mathcal{O}}(r^\omega\,d)$	$\tilde{\mathcal{O}}(r^3\,d)$
Itération de Newton	$\tilde{\mathcal{O}}(r^\omega\,d)$	$\tilde{\mathcal{O}}(r^{\omega+1}\,d)$
Storjohann	$\tilde{\mathcal{O}}(r^\omega\,d)$	$\tilde{\mathcal{O}}(r^\omega\,d)$
Notre algorithme	$\tilde{\mathcal{O}}(r^\omega + r^2\,d)$	$\tilde{\mathcal{O}}(r^3\,d)$

Tableau. Coût simplifié de la résolution de $A = B \cdot C$ pour des matrices denses sur $\Bbbk[[X]]$ avec $s = 1$.

Ensuite, nous analysons la situation pour les matrices structurées. Brièvement, dans la représentation des matrices structurées, un entier α, nommé le rang (de déplacement), est associé à toute matrice $A \in \mathcal{M}_{r\times r}(R_p)$. Les deux principales caractéristiques de la matrice structurée A est que A peut être stockée en mémoire par une structure de donnée compacte en taille $\mathcal{O}(\alpha\,r)$ et que le produit matrice-

vecteur $A \cdot V$ coûte $\mathcal{O}(\alpha\,\mathsf{M}(r))$ pour tout vecteur $V \in \mathcal{M}_{r\times 1}(R_p)$. Nous renvoyons à [Pan01] pour une présentation complète.

Le tableau qui suit rappelle les résultats de complexité connus pour la résolution de systèmes linéaires structurés et donne le temps de calcul de notre algorithme. Ici, d' désigne la longueur des entrées p-adiques de la structure de donnée compacte qui sert à stocker A. La précision voulue est toujours notée N. Comme précédemment, nous donnons des complexités simplifiées pour les séries formelles dans le cas $s=1$ et $N=d'$ ou $N=r\,d'$.

Algorithme	$N=d'$	$N=r\,d'$
Dixon	$\tilde{\mathcal{O}}\big(\alpha^2 r+\alpha\,r\,d'^2\big)$	$\tilde{\mathcal{O}}\big(\alpha\,r^2\,d'^2\big)$
Moenck-Carter	$\tilde{\mathcal{O}}(\alpha^2\,r\,d')$	$\tilde{\mathcal{O}}(\alpha\,r^2\,d')$
Itération de Newton	$\tilde{\mathcal{O}}(\alpha^2\,r\,d')$	$\tilde{\mathcal{O}}(\alpha^2\,r^2\,d')$
Notre algorithme	$\tilde{\mathcal{O}}(\alpha^2\,r+\alpha\,r\,d')$	$\tilde{\mathcal{O}}(\alpha\,r^2\,d')$

Tableau. Coût simplifié de la résolution de $A=B\cdot C$ pour des matrices structurées sur $\Bbbk[[X]]$ avec $s=1$.

Notre algorithme est le plus rapide pour des matrices structurées dans les deux cas $N=d$ et $N=r\,d$. Remarquons que l'algorithme de Moenck-Carter est aussi rapide dans le second cas.

Pour finir, nous implémentons ces algorithmes et comparons les temps de calcul pour la représentation dense. Notre implémentation est disponible dans la bibliothèque C++ nommée ALGEBRAMIX incluse dans MATHEMAGIX [HLM+02]. Comme application, nous résolvons des systèmes linéaires sur les entiers et nous nous comparons aux logiciels LINBOX et IML. Nous montrons que nous améliorons les temps pour les petites matrices à coefficients de grands entiers.

4 Séries solutions d'équations (q)-différentielles

Le but de ce chapitre est de fournir des algorithmes qui calculent les séries solutions d'une large famille d'équations, ou de systèmes, différentiels ou aux q-différences. Le nombre d'opérations arithmétiques est linéaire en la précision, à des facteurs logarithmiques près.

Nous nous concentrons sur le cas particulier des équations linéaires, puisque dans de nombreux cas une linéarisation est possible [BCO+07]. Quand l'ordre n de l'équation est strictement plus grand que 1, nous utilisons la technique classique qui transforme ces équations en des équations du premier ordre sur des vecteurs, et nous nous intéresserons ainsi à des équations de la forme

$$x^k\,\delta(F)=A\,F+C, \tag{1}$$

où A est une matrice $n\times n$ sur l'anneau des séries formelles $\Bbbk[[x]]$ ($\Bbbk$ étant le corps des coefficients), C et l'inconnue F sont des vecteurs de taille n sur $\Bbbk[[x]]$ et δ désigne temporairement l'opérateur différentiel d/dx. L'exposant k de (1) est un entier positif qui joue un rôle clé dans cette équation.

Par *résoudre* une telle équation, nous entendons calculer un vecteur F de séries formelles tel que (1) soit vrai modulo x^N. À cet effet, il est nécessaire de calculer F que comme un polynôme de degré au plus N (quand $k=0$) ou $N-1$ (autrement). Réciproquement, quand (1) a une solution série, ses premiers N coefficients peuvent être calculés en résolvant (1) modulo x^N (quand $k\neq 0$) ou x^{N-1} (autrement).

Si $k=0$ et le corps $\Bbbk$ a caractéristique 0, alors un théorème de Cauchy formel s'applique et (1) a un unique vecteur de solutions séries pour une condition initiale donnée. Dans ce cas, des algorithmes existent pour calculer les N premiers termes de la solution en complexité quasi-linéaire : [BK78] pour les équations scalaires d'ordre 1 ou 2, adapté dans [BCO+07] pour les systèmes d'équations. Les algorithmes détendus de [Hoe02] s'appliquent aussi à ce cadre.

Dans ce chapitre, nous étendons les algorithmes précédents dans trois directions.

Singularités Nous traitons le cas où k est strictement positif. Le théorème de Cauchy et les techniques de [BCO+07] ne sont pas applicables. Nous montrons dans ce chapitre comment dépasser ce comportement singulier et obtenir une complexité quasi-linéaire.

Caractéristique positive Même dans le cas $k=0$, le théorème de Cauchy ne s'applique pas en caractéristique positive et l'équation (1) peut ne pas avoir de solutions séries (un exemple simple est $F'=F$). Cependant, une telle équation peut tout de même avoir une solution modulo x^N. Nos objectifs à cet égard sont de surpasser le manque de théorème de Cauchy, ou d'une théorie formelle des équations singulières, en donnant des conditions suffisantes pour assurer l'existence de solutions à la précision demandée.

Équations fonctionnelles Les équations linéaires différentielles ou aux (q)-différences se résolvent par des algorithmes similaires. Pour ceci, introduisons un morphisme d'anneau unitaire $\sigma\colon \Bbbk[[x]]\to\Bbbk[[x]]$ et une σ-dérivation $\delta\colon \Bbbk[[x]]\to\Bbbk[[x]]$, en ce sens que δ est $\Bbbk$-linéaire et que pour tout f,g dans $\Bbbk[[x]]$, on a

$$\delta(fg)=f\delta(g)+\delta(f)\,\sigma(g).$$

Ces définitions, et l'égalité ci-dessus, s'étendent aux matrices sur $\Bbbk[[x]]$. Ainsi, notre objectif est de résoudre la généralisation suivante de (1) :

$$x^k\,\delta(F)=A\,\sigma(F)+C. \tag{2}$$

Comme auparavant, nous sommes intéressés par le fait de calculer un vecteur F de séries tel que (2) soit vrai modulo x^N.

À propos des algorithmes détendus, les techniques de [Hoe02] s'appliquent déjà en caractéristique positive. Au commencement de ma thèse, les outils pour adapter les algorithmes détendus au cas des équations singulières n'existaient pas. Notre méthode pour traiter le cas des équations singulières a été découverte indépendamment à la même époque par [Hoe11]. Ce dernier article traite d'équations récursives plus générales comme les équations algébriques, différentielles ou une combinaison des deux. Cependant, ce même article ne traite pas le cas des équations (q)-différentielles et travaille sous des hypothèses plus restrictives.

Nous nous limitons à la situation suivante :

$$\delta(x)=1,\qquad \sigma\colon x\mapsto q\,x,$$

pour un élément $q \in \Bbbk \setminus \{0\}$. Il n'y a alors que deux possibilités :

- $q = 1$ et $\delta\colon f \mapsto f'$ (cas *différentiel*) ;
- $q \neq 1$ et $\delta\colon f \mapsto \frac{f(qx) - f(x)}{x(q-1)}$ (cas (q)-*différentiel*).

En voyant l'équation (2) comme un système linéaire, on peut résoudre l'équation en utilisant des méthodes d'algèbre linéaire en dimension $n\,N$. Bien que cette solution soit toujours valable, nous donnons des algorithmes de bien meilleure complexité, sous des hypothèses de *bon spectre* liées au spectre $\operatorname{Spec} A_0$ du coefficient p-adique constant de A.

Similairement à l'article [BCO+07] pour le cas non-singulier, nous développons deux approches. La première est une méthode diviser-pour-régner. Le problème est d'abord résolu à précision $N/2$ puis le calcul à précision N est complété en résolvant un problème du même type à précision $N/2$. Cela nous donne le résultat suivant.

Théorème. *Il est possible de calculer des générateurs de l'espace des solutions de l'équation (2) à précision N par une approche diviser-pour-régner. En supposant que A_0 a un bon spectre à précision N, cela peut être fait en temps $\mathcal{O}(n^\omega\,\mathsf{M}(N)\log(N))$. Si l'on a soit $k>1$ soit $k=1$ et $q^i\,A_0 - \gamma_i\,\mathrm{Id}$ inversible pour tout $0 \leq i < N$, le temps de calcul tombe à $\mathcal{O}(n^2\,\mathsf{M}(N)\log(N) + n^\omega\,N)$.*

Cette approche diviser-pour-régner coïncide avec le calcul détendu d'un p-adique récursif sous le deuxième ensemble d'hypothèses, c'est-à-dire soit $k>1$ soit $k=1$ et $q^i\,A_0 - \gamma_i\,\mathrm{Id}$ inversible pour $0 \leq i < N$. Nous prenons le parti de le présenter comme un algorithme diviser-pour-régner car cela permettra de traiter plus facilement le problème sous des hypothèses plus générales.

Notre deuxième algorithme se comporte mieux par rapport à N, avec un coût de seulement $\mathcal{O}(\mathsf{M}(N))$, mais il nécessite des multiplications de matrices polynomiales. Comme dans de nombreux cas, l'approche diviser-pour-régner évite de telles multiplications, le deuxième algorithme est à préférer pour des précisions relativement grandes.

Dans le cas différentiel, quand $k=0$ et que la caractéristique est 0, les algorithmes de [BCO+07, BK78] calculent une matrice inversible de solutions séries de l'équation homogène par une itération de Newton puis en déduisent une solution du problème initial par la méthode de la variation de la constante. Dans le contexte plus général que nous considérons ici, une telle matrice n'existe pas. Cependant, il se trouve qu'une équation associée dérivée de (2) admet une telle solution. Nous utilisons alors une variante de l'itération de Newton pour résoudre cette équation et obtenons le résultat suivant.

Théorème. *En supposant que A_0 a un bon spectre à précision N, il est possible de calculer des générateurs de l'espace des solutions de l'équation (2) à précision N par une itération semblable à celle de Newton en temps $\mathcal{O}(n^\omega\,\mathsf{M}(N) + n^\omega \log(n)\,N)$.*

À notre connaissance, c'est la première fois qu'une complexité aussi basse est atteinte pour ce problème. Cependant, si l'on retire l'hypothèse de bon spectre, nous ne pouvons plus garantir que l'algorithme est correct, et encore moins contrôler sa complexité.

5 Remontée détendue pour les systèmes algébriques

Ce chapitre peut être vu comme un cas particulier de la remontée d'ensemble triangulaire faite dans le chapitre 6.

Supposons qu'il nous est donné un système polynomial $\boldsymbol{P} = (P_1, ..., P_r)$ dans $R[Y_1, ..., Y_r]$ ainsi que $\boldsymbol{y}_0 \in (R/(p))^r$ tel que $\boldsymbol{P}(\boldsymbol{y}_0) = 0$ dans $(R/(p))^r$. Nous travaillerons sous l'hypothèse du Lemme de Hensel qui requiert que la matrice jacobienne $\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)$ soit inversible. En conséquence, il existe une unique racine $\boldsymbol{y} \in R_p^r$ de $\boldsymbol{P}$ qui se réduise à $\boldsymbol{y}_0$ modulo p.

Notre travail consiste à transformer ces équations implicites en des équations récursives. Alors, nous pourrons utiliser le cadre des p-adiques récursifs détendus pour remonter une racine résiduelle $\boldsymbol{y}_0 \in (R/(p))^r$ en la racine $\boldsymbol{y} \in (R_p)^r$ de $\boldsymbol{P}$ sur l'anneau des p-adiques. Nos résultats sur la transformation d'équations implicites en équations récursives furent découverts en même temps par [Hoe11].

Par souci de clarté, nous ne donnons ici que les complexités asymptotiques quand la précision N des p-adiques tend vers l'infini. Par conséquent, nous noterons $f(n, L, d, N) = \mathcal{O}_{N\to\infty}(g(n, L, d, N))$ s'il existe $K_{n,L,d} \in \mathbb{R}_{\geqslant 0}$ tel que pour tout $N \in \mathbb{N}$, on ait $f(n, L, d, N) \leqslant K_{n,L,d}\, g(n, L, d, N)$.

Commençons par énoncer le résultat pour les polynômes denses à une variable.

Proposition. *Étant donné un polynôme P de degré d en représentation dense et une racine simple modulaire y_0 de P, la remontée de la racine y_0 à précision N dans R_p coûte $(d-1)\,\mathsf{R}(N) + \mathcal{O}_{N\to\infty}(N)$.*

En comparaison, l'itération de Newton remonte y à précision N en temps $(3\,d + 4)\,\mathsf{I}(N) + \mathcal{O}_{N\to\infty}(N)$ (*cf.* [GG03, Théorème 9.25]). Le premier avantage de notre algorithme en-ligne est qu'il fait moins de multiplications en-ligne que l'algorithme hors-ligne ne fait de multiplications hors-ligne. Du coup, nous pouvons espérer des temps de calcul meilleurs pour l'algorithme en-ligne quand la précision N est telle que $\mathsf{R}(N) \leqslant 3\,\mathsf{I}(N)$.

Traitons maintenant le cas des systèmes de polynômes à plusieurs variables.

Théorème. *Soient $\boldsymbol{P} = (P_1, ..., P_r)$ un système polynomial dans $R[Y_1, ..., Y_r]^r$ donné en représentation dense, vérifiant $d \geqslant 2$ avec $d := \max_{1\leqslant i,j\leqslant r}(\deg_{X_j}(P_i)) + 1$, et $\boldsymbol{y}_0$ un zéro résiduel de $\boldsymbol{P}$ sur $R/(p)$.*

Alors on peut calculer $\boldsymbol{y}$ à précision N en temps $d^r\,\mathsf{R}(N) + \mathcal{O}_{N\to\infty}(N)$.

Comparons-nous à nouveau avec l'itération de Newton qui nécessite à chaque étape une évaluation des équations polynomiales et de leur matrice jacobienne, et une inversion de la matrice jacobienne évaluée. Cela coûte $C\,(r\,d^r + r^\omega)\,\mathsf{I}(N) + \mathcal{O}_{N\to\infty}(N)$ où C est une constante réelle strictement positive. Le théorème précédent montre que l'on peut économiser le coût de l'inversion de la matrice jacobienne à précision N avec les algorithmes en-ligne.

Cet avantage est d'autant plus significatif que le coût de l'évaluation du système est inférieur au coût de l'inversion de la matrice. Pour mieux quantifier ce phénomène, nous adaptons notre approche détendue aux polynômes donnés par des calculs d'évaluation directs (« straight-line programs » en anglais), c'est-à-dire donnés comme une suite d'opérations arithmétiques sans branchement.

Théorème. *Soit $\boldsymbol{P}$ un système polynomial de r polynômes à r variables sur R, donné par un calcul d'évaluation direct contenant L^* multiplications. Soit $\boldsymbol{y}_0 \in (R/(p))^r$ tel que $\boldsymbol{P}(\boldsymbol{y}_0) = 0 \bmod p$ et $\det(\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)) \neq 0 \bmod p$.*

Alors, le calcul de $\boldsymbol{y}$ à précision N coûte $3\,L^*\,\mathsf{R}(N) + \mathcal{O}_{N\to\infty}(N)$.

Dans ce cas, l'itération de Newton coûte $C'\,(L^* + r^\omega)\,\mathsf{I}(N) + \mathcal{O}_{N\to\infty}(N)$, où C' est une autre constante réelle positive. Ainsi, notre approche détendue est particulièrement adaptée aux systèmes polynomiaux qui s'évaluent bien, comme par exemple les systèmes creux. Notons que malgré ces avantages, nos algorithmes sont plus coûteux d'un facteur logarithmique en la précision N par rapport à l'itération de Newton.

Finalement, nous implantons ces algorithmes et obtenons des temps de calculs compétitifs avec Newton, et même meilleurs sur une grande plage de paramètres. Notre implémentation est disponible dans la bibliothèque C++ ALGEBRAMIX de MATHEMAGIX [HLM+02].

6 Remontée détendue d'ensembles triangulaires

De la même manière que l'opérateur de Newton-Hensel a été adapté pour remonter des représentations à une variable dans [GLS01, HMW01] puis des représentations triangulaires dans [Sch02], nous adaptons notre approche détendue du chapitre 5 pour remonter de tels objets. Aussi, nous avions remarqué en section 5 que les algorithmes détendus peuvent économiser le coût de l'algèbre linéaire pour la remontée de racines de systèmes polynomiaux, *a contrario* des méthodes hors-ligne. Nous souhaitons aussi économiser ce coût dans le cadre présent.

Introduisons la notion de représentation à une variable d'un idéal zéro-dimensionnel $\mathcal{I} \subseteq R[X_1, \ldots, X_n]$. Soient A l'algèbre quotient $R[X_1, \ldots, X_n]/\mathcal{I}$ et $\Lambda \in A$ tels que la R-algèbre $R[\Lambda]$ engendrée par Λ est égale à A. Une *représentation à une variable* de A est une famille de polynômes $\mathfrak{P} = (Q, S_1, \ldots, S_n)$ de $R[T]$ avec $\deg(S_i) < \deg(Q)$ telle que l'on ait l'isomorphisme de R-algèbre suivant

$$\begin{array}{ccc} A = R[X_1, \ldots, X_n]/\mathcal{I} & \to & R[T]/(Q) \\ X_1, \ldots, X_n & \mapsto & S_1, \ldots, S_n \\ \Lambda & \mapsto & T. \end{array}$$

La trace la plus ancienne de cette représentation se trouve dans [Kro82] et quelques années plus tard dans [Kön03]. L'article [Mac16] contient un bon résumé de leur travaux. Le « shape lemma » de [GM89] énonce l'existence d'une telle représentation pour une forme linéaire générique Λ d'un idéal zéro-dimensionnel. Il existe tout une famille d'algorithmes pour calculer cette représentation, utilisant une résolution géométrique [GHMP97, GHH+97, GLS01, HMW01] ou des bases de Gröbner [Rou99].

Un *ensemble triangulaire* est une famille de n polynômes $\boldsymbol{t} = (t_1, \ldots, t_n)$ de $R[X_1, \ldots, X_n]$ telle que t_i est dans $R[X_1, \ldots, X_i]$, unitaire en X_i et réduit par rapport à $(t_1, \ldots, t_{i-1})$. La notion d'ensemble triangulaire est née avec l'article [Rit66] dans le contexte des algèbres différentielles. Plusieurs notions similaires ont été introduites après coup dans les articles [Wu84, Laz91, Kal93, ALMM99]. Bien que ces notions ne coïncident pas en général, elles définissent le même objet dans le cas des idéaux zéro-dimensionnels.

Les représentations à une variable peuvent être vues comme un cas particulier d'ensemble triangulaire. En effet, avec les notations précédentes, la famille $(Q(T), X_1 - S_1(T), ..., X_n - S_n(T))$ est un ensemble triangulaire de l'algèbre $R[T, X_1, ..., X_n]$. À partir de maintenant, nous considérerons les représentations à une variable comme un cas particulier d'ensembles triangulaires.

Définissons $\mathsf{Rem}(d_1, \ldots, d_n)$ comme étant le coût d'une opération arithmétique dans l'algèbre quotient $R[X_1, ..., X_n]/(t_1, ..., t_n)$ où $d_i := \deg_{X_i}(t_i)$. Quand on utilise une représentation à une variable, les éléments de $A \simeq R[T]/(Q)$ sont représentés comme des polynômes à une variable de degré strictement inférieur à $d := \deg(Q)$. Du coup, la multiplication dans A coûte quelques multiplications polynomiales.

La remontée d'ensembles triangulaires (ou de représentation à une variable) est une opération cruciale. Plusieurs implémentations d'algorithmes qui calculent des ensembles triangulaires sur les rationnels se ramènent à calculer ces objets modulo un nombre premier, puis à faire une remontée p-adique et enfin une reconstruction rationnelle. Par exemple, le logiciel Kronecker [L+02] qui calcule des représentations à une variable, et la bibliothèque RegularChains [LMX05] de Maple qui calcule des ensembles triangulaires, utilisent des remontées p-adiques. Mieux encore, une autre remontée est utilisée au cœur de l'algorithme de résolution géométrique [GLS01, HMW01] qui sous-tend Kronecker. En effet, cet algorithme manipule des représentations à une variable de courbes et nécessite donc des remontées sur les séries formelles.

En pratique, la majorité du temps de calcul d'un ensemble triangulaire est passé dans la remontée. Donc toute amélioration de la complexité de la remontée impactera directement l'algorithme complet.

Soit $\boldsymbol{f} = (f_1, \ldots, f_n) \in R[X_1, \ldots, X_n]$ un système polynomial donné par un calcul d'évaluation direct avec L opérations arithmétiques. Si L_{f_i} est le nombre d'opérations nécessaires juste pour la sortie f_i, alors nous notons $L^\perp := L_{f_1} + \cdots + L_{f_n}$ la complexité des calculs indépendants de $f_1, \ldots, f_n$, c'est-à-dire sans partage de calculs entre eux. Puisque $L_{f_i} \leqslant L$, nous avons

$$L \leqslant L^\perp \leqslant n\, L.$$

Un algorithme dû à Baur et Strassen [BS83] permet d'évaluer la matrice jacobienne de $\boldsymbol{f}$ par un calcul direct en $5\, L^\perp$ opérations.

Soit $\boldsymbol{t}_0$ un ensemble triangulaire de $R/(p)[X_1, ..., X_n]$ tel que $\boldsymbol{f} = 0$ dans l'algèbre $R/(p)[X_1, ..., X_n]/\langle \boldsymbol{t}_0 \rangle$. Nous travaillerons sous l'hypothèse que le déterminant de la matrice jacobienne $\mathrm{Jac}_{\boldsymbol{f}}$ dans $\mathcal{M}_n(R/(p)[X_1, ..., X_n])$ est inversible modulo $\boldsymbol{t}_0$. Cette condition est suffisante pour avoir l'existence et l'unicité d'un ensemble triangulaire $\boldsymbol{t}$ de $R_p[X_1, ..., X_n]$ qui se réduit à $\boldsymbol{t}_0$ modulo p et satisfait $\boldsymbol{f} = 0$ dans $R_p[X_1, ..., X_n]/\langle \boldsymbol{t} \rangle$.

L'algorithme de remontée calcule, à partir des entrées $\boldsymbol{f}$ et $\boldsymbol{t}_0$, cet unique ensemble triangulaire $\boldsymbol{t}$ à précision N.

Notre contribution consiste à donner, pour tout ensemble triangulaire p-adique, un algorithme décalé dont il est point fixe. Nous appliquons alors le cadre détendu pour la remontée de p-adiques récursifs et en déduisons un algorithme pour calculer cet ensemble triangulaire.

Par souci de clarté, nous ne donnerons que les complexités asymptotiques en la précision N des p-adiques. Énonçons les résultats de complexité.

Théorème. *Notre algorithme détendu peut effectuer la remontée de l'ensemble triangulaire* $\boldsymbol{t}$ *à précision* N *en temps*

$$C\,n\,L\,\mathsf{R}(N)\,\mathsf{Rem}(d_1,...,d_n)+\mathcal{O}_{N\to\infty}(N),$$

avec C *une constante réelle positive.*

Le précédent algorithme hors-ligne de [Sch02] fait la remontée de $\boldsymbol{t}$ à précision N en temps $C'\,(L^{\perp}+n^{\omega})\,\mathsf{I}(N)\,\mathsf{Rem}(d_1,...,d_n)+\mathcal{O}_{N\to\infty}(N)$ avec C' une autre constante réelle positive. Notre algorithme détendu n'améliore donc la complexité par rapport au précédent algorithme que dans le cas $n\,L\leqslant n^{\omega}$, c'est-à-dire pour les systèmes $\boldsymbol{f}$ qui s'évaluent en moins de n^2 opérations, en prenant $\omega=3$.

La situation est plus à notre avantage pour la remontée de représentations à une variable. Supposons que $\boldsymbol{t}$ est une représentation à une variable de degré d.

Théorème. *Notre algorithme détendu peut remonter la représentation à une variable* $\boldsymbol{t}$ *à précision* N *en temps*

$$C''\,L\,\mathsf{R}(N)\,\mathsf{M}(d)+\mathcal{O}_{N\to\infty}(N),$$

avec C'' *une constante réelle positive.*

Comparons notre algorithme avec l'algorithme hors-ligne existant de [GLS01, HMW01]. Cet algorithme hors-ligne remonte une représentation à une variable $\boldsymbol{t}$ à précision N en temps $C'''\ (L^{\perp}+n^{\omega})\ \mathsf{I}(N)\ \mathsf{M}(d)+\mathcal{O}_{N\to\infty}(N)$, où C''' est une autre constante strictement positive. Par conséquent, notre algorithme détendu fait toujours asymptotiquement moins de multiplications en-ligne que l'autre algorithme ne fait de multiplications hors-ligne. De plus, pour les systèmes polynomiaux $\boldsymbol{f}$ qui s'évaluent en moins de n^{ω} opérations, nous pouvons nous attendre à un gain de performance significatif de la part de notre algorithme.

Pour finir, nous présentons une implémentation de cet algorithme au sein de la bibliothèque C++ ALGEBRAMIX de MATHEMAGIX [HLM+02] dans le cas particulier des représentations à une variable. Notre algorithme se compare favorablement sur les exemples présentés. Mentionnons aussi que notre algorithme détendu est en cours de branchement à KRONECKER dans MATHEMAGIX avec l'aide de G. Lecerf.

7 Algorithmique de l'algèbre de décomposition universelle

Soient $\Bbbk$ un corps et $f\in\Bbbk[X]$ un polynôme séparable de degré n. Notons $\mathcal{R}:=\{\alpha_1,...,\alpha_n\}$ l'ensemble des racines de f dans une clôture algébrique de $\Bbbk$. *L'idéal des relations symétriques* $\mathcal{I}_s$ est l'idéal

$$\{P\in\Bbbk[X_1,...,X_n]\,|\,\forall\sigma\in\mathfrak{S}_n,P(\alpha_{\sigma(1)},...,\alpha_{\sigma(n)})=0\}.$$

L'algèbre de décomposition universelle est l'algèbre quotient $\mathbb{A}:=\Bbbk[X_1,...,X_n]/\mathcal{I}_s$, qui est zéro-dimensionnelle de degré $\delta:=n!$.

Le but de ce chapitre est de calculer efficacement dans $\mathbb{A}$. Nous utilisons une représentation à une variable de $\mathbb{A}$, c'est-à-dire un isomorphisme de la forme $\mathbb{A} \simeq \Bbbk[T]/Q(T)$, puisque les opérations arithmétiques dans $\mathbb{A}$ ont un coût quasi-optimal dans cette représentation. Nous détaillerons deux algorithmes liés, l'un pour calculer l'isomorphisme précédent et l'autre pour calculer le polynôme caractéristique d'un élément de $\mathbb{A}$. Ces algorithmes sont les premiers à être quasi-optimaux pour ces tâches.

Nous mesurerons le coût de nos algorithmes par le nombre d'opérations arithmétiques dans $\Bbbk$ qu'ils font. En pratique, ce modèle de complexité est bien adapté au cas où $\Bbbk$ est un corps fini ; sur $\Bbbk = \mathbb{Q}$, il conviendrait d'utiliser une remontée p-adique comme celle du chapitre 6.

La question de quelle représentation doit être utilisée pour $\mathbb{A}$ est le cœur de l'article ainsi que la clé pour obtenir de meilleurs algorithmes. Une représentation répandue est celle par *ensembles triangulaires*. Les *différences divisées*, aussi connues sous le nom de *modules de Cauchy* [Che50, RV99], sont définies par $C_1(X_1) := f(X_1)$ et récursivement

$$C_{i+1} := \frac{C_i(X_1, ..., X_i) - C_i(X_1, ..., X_{i-1}, X_{i+1})}{X_i - X_{i+1}}$$

pour $1 \leqslant i < n$. Elles forment une *base triangulaire* de $\mathcal{I}_s$. Les différences divisées sont peu coûteuses à calculer à partir de leur formule récursive mais il est difficile de rendre les calculs dans $\mathbb{A}$ efficace dans cette représentation. L'article [BCHS11] donne un algorithme de multiplication qui coûte $\tilde{\mathcal{O}}(\delta)$, mais cet algorithme cache des facteurs logarithmiques de hauts degrés dans le grand-O. Il n'y a pas d'algorithme connu pour l'inversion d'éléments de $\mathbb{A}$ qui soit quasi-linéaire.

Le seconde représentation que nous considérerons est celle à une variable. Dans cette représentation, les éléments de $\mathbb{A} \simeq \Bbbk[T]/(Q)$ sont représentés comme des polynômes à une variable de degré strictement inférieur à δ. La multiplication et l'inversion (si possible) dans $\mathbb{A}$ coûte alors respectivement $\mathcal{O}(\mathsf{M}(\delta))$ et $\mathcal{O}(\mathsf{M}(\delta)\log(\delta))$. Pour le polynôme caractéristique, la situation n'est pas aussi favorable puisqu'aucun algorithme quasi-linéaire n'est connu : le meilleur résultat, dû à [Sho94], est $\mathcal{O}(\mathsf{M}(\delta)\ \delta^{1/2} + \delta^{(\omega+1)/2})$, ce qui donne un algorithme en $\mathcal{O}(\delta^{1.69})$ opérations pour le polynôme caractéristique.

Le calcul d'une représentation à une variable de $\mathbb{A}$ est coûteux : le meilleur algorithme existant, dû à [PS11], prend en entrée un ensemble triangulaire (tel que les différences divisées) et le convertit en une représentation à une variable en temps $\tilde{\mathcal{O}}(\delta^{1.69})$.

Pour résumer, une représentation triangulaire de $\mathbb{A}$ est facile à calculer mais implique une algorithmique plutôt inefficace pour $\mathbb{A}$. D'un autre côté, le calcul d'une représentation à une variable n'est pas chose aisée, mais une fois qu'il est réalisé, certains calculs dans $\mathbb{A}$ deviennent plus rapides. La principale contribution de ce chapitre est de montrer comment contourner les inconvénients des représentations à une variable, en donnant des algorithmes rapides pour leur construction. Nous expliquons aussi comment utiliser l'arithmétique rapide à une variable dans $\mathbb{A}$ pour calculer efficacement des polynômes caractéristiques.

Dans le cas des représentations à une variable, nos algorithmes sont Las Vegas et nous donnons alors la complexité moyenne des algorithmes.

Théorème. *Supposons que la caractéristique de* $\Bbbk$ *est zéro, ou supérieure à* $2\,\delta^2$. *Alors nous pouvons calculer une représentation à une variable de* $\mathbb{A}$ *et des polynômes caractéristiques dans les temps indiqués dans le tableau suivant.*

$\mathcal{X}_{P,\mathbb{A}}$	*représentation à une variable (temps moyen)*
$\mathcal{O}(n^{(\omega+3)/2}\,\mathsf{M}(\delta)) = \tilde{\mathcal{O}}(\delta)$	$\mathcal{O}(n^{(\omega+3)/2}\,\mathsf{M}(\delta)) = \tilde{\mathcal{O}}(\delta)$

Nous proposons deux approches qui sont toutes les deux basées sur des idées classiques. La première approche calcule des polynômes caractéristiques grâce à leurs sommes de Newton, à la suite des travaux [Val89, AV94, CM94], mais en se limitant à des polynômes simples, comme par exemple les formes linéaires. Cela produira le meilleur algorithme en pratique. La deuxième approche se base sur des résultants itérés [Lag70, Soi81, Leh97, RV99] et fournit les estimations de complexité du théorème.

Finalement, nous implémentons nos algorithmes dans MAGMA 2.17.1 [BCP97] et donnons des résultats expérimentaux. Nous observons des améliorations pratiques pour le calcul de représentation à une variable de $\mathbb{A}$. Notre algorithme de changement de base entre des représentations à une variable et triangulaires est efficace ; pour une opération telle que l'inversion, et malgré le surcoût dû au changement de représentation, il est plus efficace de passer par une représentation à une variable.

8 Remontée d'invariants fondamentaux

Ce court appendice est dédié à la preuve d'un résultat utile en théorie des invariants que nous avons obtenu lors de la rédaction du chapitre 7 : nous montrons que les invariants fondamentaux de l'action d'un groupe fini se spécialisent toujours bien modulo tout nombre premier, sauf un petit nombre connu à l'avance. Ce phénomène est rare en Calcul Formel puisque empiriquement, pour des systèmes non-linéaires, les nombres premiers donnant lieu à une mauvaise réduction ne peuvent pas être déterminés directement.

Ce résultat a des implications pratiques : pour calculer les invariants fondamentaux sur les rationnels, il suffit de les calculer modulo p. La remontée vers les rationnels est alors triviale.

Une correspondance privée avec H. E. A. Campbell, D. Wehlau et M. Roth nous a appris que ce résultat leur était déjà connu mais, à notre connaissance, jamais publié. Nous décidons de l'inclure dans cette thèse pour qu'il puisse être utilisé en pratique : à notre connaissance, un logiciel tel que MAGMA [BCP97] n'utilise pas ce résultat pour calculer des anneaux d'invariants.

Partie I

On-line algorithms

Chapitre 1
Relaxed algorithms for multiplication

This chapter introduces the notions of online and relaxed algorithms. First, we present a general context of p-adic computations that will be in use for the next few chapters. Then, we recall the current relaxed algorithms for the multiplication, and we give for the first time a thorough analysis of their arithmetic complexity. In a third time, we introduce a new relaxed algorithm for the multiplication using *middle* and *short* product, that improves by a constant factor the previous relaxed multiplication. Finally, we give some timings to confirm the good behavior of relaxed algorithms with middle product.

1.1 Computing with p-adics

This section introduces several important notions and notation regarding p-adic computations, which will be in use for the next few chapters.

1.1.1 Basic definitions

Let R be a commutative ring with unit. We consider an element $p \in R - \{0\}$, and we write R_p for the completion of the ring R for the p-adic valuation. We will assume that $R/(p)$ is a field (equivalently, that p generates a maximal ideal). This is not needed for the algorithms in this chapter, but will be useful later on when we deal with linear algebra modulo (p). We also assume that $\cap_{i\in\mathbb{N}}(p^i) = \{0\}$, so that R can be seen as a subset of R_p.

An element $a \in R_p$ is called a *p-adic*; it can always be written (in a non unique way) $a = \sum_{i\in\mathbb{N}} a_i p^i$ with coefficients $a_i \in R$.

To get a unique representation of elements in R_p, we will fix a subset M of R such that the projection $\pi\colon M \to R/(p)$ is a bijection. Then, any element $a \in R_p$ can be uniquely written $a = \sum_{i\in\mathbb{N}} a_i p^i$ with coefficients $a_i \in M$. The operations mod and quo are then defined as

$$a \bmod p = a_0 \quad \text{and} \quad a \operatorname{quo} p = \sum_{i>0} a_i p^{i-1}.$$

We will suppose that for all $a \in M$, $-a$ is in M as well.

Two classical examples are the formal power series ring $\Bbbk[[X]]$, which is the completion of the ring of polynomials $\Bbbk[X]$ for the ideal (X), and the ring of p-adic integers $\mathbb{Z}_p$, which is the completion of the ring of integers $\mathbb{Z}$ for the ideal (p), with p a prime number. For $R = \Bbbk[X]$, we naturally take $M = \Bbbk$; for $R = \mathbb{Z}$, we choose $M = \left\{-\frac{p-1}{2}, \ldots, \frac{p-1}{2}\right\}$ if p is odd and $M = \{0, 1\}$ for $p = 2$.

Once M has been fixed, we have a well-defined notion of *length* of a (non-zero) p-adic: if $a = \sum_{i\in\mathbb{N}} a_i\, p^i$, then we define

$$\lambda(a) := 1 + \sup\,(i \in \mathbb{N} \,|\, a_i \neq 0),$$

so that $\lambda(a)$ is in $\mathbb{N}_{>0} \cup \{\infty\}$; for $a = 0$, we take $\lambda(a) = 0$. Since M is invariant through sign change, we have that $\lambda(-a) = \lambda(a)$ for all a. We will make the following assumptions:

- λ verifies the conditions

$$\begin{aligned} \lambda(a+b) &\leqslant \max\,(\lambda(a), \lambda(b)) + 1 \\ \lambda(a\,b) &\leqslant \lambda(a) + \lambda(b); \end{aligned}$$

- all elements of $R \subset R_p$ have finite length (this excludes cases where for instance R is already complete with respect to the (p)-adic topology).

These assumptions are satisfied in the two main cases above (with further simplifications in the polynomial case, since no carries appear in the case of addition); note that $\lambda(a-b)$ satisfies the same inequality as $\lambda(a+b)$.

For any $a \in R_p$ and integers $0 \leqslant r \leqslant s$, we define the truncated p-adic $a_{r\ldots s}$ as

$$a_{r\ldots s} := a_r + a_{r+1}\, p + \cdots + a_{s-1}\, p^{s-1-r} \in R.$$

We call *p-adics at precision n* the set of all truncations $a_{0\ldots n}$ of p-adics $a \in R_p$ (for the two main cases we have in mind, they are simply plain integers, resp. polynomials). We say that we have computed a p-adic at precision n if the result holds modulo p^n.

1.1.2 Basic operations

Algorithmically, we represent p-adics through their base-M expansion, that is, through a sequence of coefficients in M. Roughly speaking, we measure the cost of an algorithm by the number of arithmetic operations with operands in M it performs. More precisely, we assume that we can do the following at unit cost:

- given a_0, b_0 in M, compute the coefficients c_0, c_1 of $c = a_0\, b_0$ at unit cost, and similarly for the coefficients of $a \pm b$
- given a_0 in $M - \{0\}$, compute b_0 in $M - \{0\}$ such that $a_0\, b_0 = 1 \bmod p$

Remark that when $R = \Bbbk[X]$, we are simply counting arithmetic operations in $\Bbbk$.

The main operations we will need on p-adics are sum and difference, as well as multiplication and a few of its variants (of course, these algorithms only operate on truncated p-adics). Addition (and subtraction) are easy to deal with:

Lemma 1.1. *The following holds:*

- *Given two p-adics a, b of length at most ℓ, one can compute $a+b$ in time $\mathcal{O}(\ell)$*
- *Given p-adics $a_1, \ldots, a_N$ of length at most ℓ, the p-adic $A = \sum_{i=1}^N a_i$ has length $\mathcal{O}(\log(N) + \ell)$, and one can compute it in time $\mathcal{O}(N\,\ell)$.*
- *Given p-adics $a_1, \ldots, a_N$ of length at most ℓ, the p-adic $A = \sum_{i=1}^N a_i\, p^i$ has length $\mathcal{O}(N + \ell)$, and one can compute it in time $\mathcal{O}(N\ell)$.*

Proof. The first point is easily dealt with by induction on ℓ; we will see the algorithm in more detail in Example 1.5 below. To handle the second one, we build a tree adder, which has depth $\mathcal{O}(\log(N))$. The length bound follows; the complexity bound comes from noticing that we do $\mathcal{O}(N)$ additions of p-adics of length ℓ, $\mathcal{O}(N/2)$ additions of p-adics of length $\ell+1$, $\mathcal{O}(N/4)$ additions of p-adics of length $\ell+2$, etc.

To deal with the last point, note that for all i, $a_i\, p^i$ has length at most $\ell + N$. Using the second point, we deduce the length bound, and the upper bound $\mathcal{O}(N\ (\ell + N))$ on the time it takes to compute the sum. If $N \leqslant \ell$, we are done. Else, we rewrite the sum as $\sum_{j=0}^{\ell-1} b_j\, p^j$, where b_j is the p-adic of length N whose coefficients are the coefficients of index j of $a_1, \ldots, a_N$. Thus, we have reversed the roles of ℓ and N, so the claim is valid in all cases. □

For multiplication, we will distinguish several variants; for the moment, we will simply define the problems, and introduce notation for their complexity.

First, we consider "plain" multiplication: given a and b of length at most n, compute their product (which has length at most $2n$). For this operation, we will let $\mathsf{I}\colon \mathbb{N} \to \mathbb{N}$ be such that all coefficients of $a\, b$ can be computed in $\mathsf{I}(n)$ operations. We will assume that $\mathsf{I}(n)$ satisfies the property that $\mathsf{I}(n)/n$ is non-decreasing. Using Fast Fourier Transform, it is possible to take $\mathsf{I}(n)$ *quasi-linear*, that is, linear up to logarithmic factors: we will review this in the next section.

Two related problems will be of interest: *short* and *middle* products. The short product at precision n is essentially the product of p-adics modulo p^n; precisely, on input a and b with $\max(\lambda(a), \lambda(b)) = n$, it computes the coefficients of

$$\mathrm{SP}(a, b) \ := \sum_{0 \leqslant i+j < n} a_i\, b_j\, p^{i+j}.$$

The definition of the middle product is slightly more complex: if a and b are p-adics with $\lambda(b) = n$, the middle product of a and b is defined as (essentially) the middle part of the product $c := a\, b$; precisely, it computes

$$\mathrm{MP}(a, b) \ := \sum_{n-1 \leqslant i+j \leqslant 2n-2} a_i\, b_j\, p^{i+j}.$$

In general, attention must be paid to carries: because of them, $\mathrm{MP}(a, b)$ may not consist in exactly the middle coefficients of $a\, b$. In the case where $R = \Bbbk[X]$, though, middle and short products simply compute a few of the coefficients of the product $a\, b$, so they can be computed by means of "plain" multiplication algorithms. We will see below that savings are possible: Section 1.2 gives algorithms for short and middle products, with a focus on the important particular case where $R = \Bbbk[X]$.

1.1.3 On-line and relaxed algorithms

Next, we introduce the "relaxed" model of computation for p-adics. Although this terminology is recent and was introduced in [Hoe02], it bears upon older and more general notions of lazy and on-line algorithms.

To the best of our knowledge, the notion of on-line Turing machine comes from [Hen66]. We give the definition formulated in [FS74].

Definition 1.2. ([Hen66, FS74]) *Let us consider a Turing machine which computes a function f on sequences, where $f\colon \Sigma^* \times \Sigma^* \to \Delta^*$, Σ and Δ are sets. The machine is said to compute f* on-line *if for all input sequences $a = a_0 a_1 \ldots a_n$, $b = b_0 b_1 \ldots b_n$ and corresponding outputs $f(a,b) = c_0 c_1 \ldots c_n$, with $a_i, b_j \in \Sigma$, $c_k \in \Delta$, it produces c_k before reading either a_j or b_j for $0 \leqslant k < j \leqslant n$.*

The machine computes f half-line *(with respect to the first argument) if it produces c_k before reading a_j for $0 \leqslant k < j \leqslant n$. The input a will be referred to as the* on-line argument *and b as the* off-line argument*.*

This definition can easily be adapted to more inputs and outputs by changing the sets Σ and Δ.

Lazy algorithms for power series are the adaptation of the *lazy evaluation* (also known as call-by-need) function evaluation scheme to computer algebra [Kar97], whose principle is to delay as much as possible the evaluation of the argument of a function. In the lazy approach, power series are represented as streams of coefficients, and the expressions they are involved in are evaluated as soon as the needed coefficients are provided; for this reason, algorithms in the lazy framework are on-line algorithms. Therefore, we will use the following informal definition.

Definition 1.3. Lazy algorithms *are on-line algorithms that try to minimize the cost at each step.*

Relaxed algorithms are also on-line algorithms. In opposition to lazy algorithms, they can do more computations at some step in order to anticipate future computations. Therefore we will use the following informal definition.

Definition 1.4. Relaxed algorithms *are on-line algorithms that try to minimize the asymptotic cost.*

Semi-relaxed algorithms are the counterpart of half-line algorithms. Although these notions were introduced for power series at first, they are easy to extend to any p-adic ring R_p.

The next chapters of this thesis present a fundamental application of relaxed algorithms, the computation of *recursive* p-adics. Meanwhile, we give some examples of on-line algorithms for two basic operations, sum and product. They are both based on an incremental process, which outputs one coefficient at a time.

Example 1.5. The first example of an on-line algorithm is the addition of p-adics. For computing the addition of p-adics a and b, we use a subroutine that takes as input another $c \in R_p$ that stores the current state of the computation and an integer i for the step of the computation we are at.

Algorithm `LazyAddStep`
Input: $a, b, c \in R_p$ and $i \in \mathbb{N}$ **Output:** $c \in R_p$
1. $c = c + (a_i + b_i)\, p^i$
2. **return** c

The addition algorithm itself follows:

Algorithm `LazyAdd`

Input: $a, b \in R_p$ and $n \in \mathbb{N}$
Output: $c \in R_p$ such that $c = (a+b) \bmod p^{n+1}$

1. $c = 0$
2. **for** i from 0 to n
   a. $c = \texttt{LazyAddStep}(a, b, c, i)$
3. **return** c

This addition algorithm is on-line: it outputs the coefficient c_i of the addition $c = a + b$ without using any a_j or b_j of index $j > i$. After each step i, c represents the sum of $a \bmod p^{i+1}$ and $b \bmod p^{i+1}$; thus, it has length at most $i + 2$. As a result, at every step, we are simply computing $a_i + b_i + c_i$ (which we know has length at most 2), and insert the result in c_i and possibly c_{i+1}.

This algorithm is also lazy as it does only the minimal number of arithmetic operations at each step. Algorithm `LazyAdd` is relaxed because it does $\mathcal{O}(n)$ additions of length-1 p-adics, which is essentially optimal, to compute the addition of two p-adics at precision n. One can write an algorithm `LazySub` similarly.

Example 1.6. Let us next present the naive on-line algorithm for multiplication of p-adics.

Algorithm `LazyMulStep`

Input: $a, b, c \in R_p$ and $i \in \mathbb{N}$
Output: $c \in R_p$

1. $c = c + \left(\sum_{j=0}^{i} a_j b_{i-j}\right) p^i$
2. **return** c

Algorithm `LazyMul`

Input: $a, b \in R_p$ and $n \in \mathbb{N}$
Output: $c \in R_p$ such that $c = (a\,b) \bmod p^{n+1}$

1. $c = 0$
2. **for** i from 0 to n
   a. $c = \texttt{LazyMulStep}(a, b, c, i)$
3. **return** c

Algorithm `LazyMul` is on-line because it outputs c_i without reading the coefficients a_j and b_j of the inputs for $j > i$. It is a lazy algorithm because it computes no more that $(a\,b)_i$ at step i. It allows the multiplication of two p-adics at precision n at cost $\mathcal{O}(n^2)$.

However, the cost of Algorithm `LazyMul` is prohibitive compared to the quasi-linear algorithms for the multiplication of high-order p-adics: this algorithm is not relaxed.

On the other hand, most fast algorithms for multiplication, such as those based on Fourier Transform, are not on-line. We remedy to this fact in Section 1.3 by presenting quasi-linear time on-line multiplication algorithms (which will thus be called *relaxed*).

1.2 Off-line multiplication

In this section, we review some existing off-line multiplication algorithms (for the plain, short and middle product), with a focus on the case where $R = \Bbbk[X]$. In the papers of van der Hoeven, off-line algorithms are also called *zealous* algorithms.

As customary, let us denote by $\mathsf{M}(n)$ a function such that over any ring, polynomials of degree at most $n-1$ can be multiplied in $\mathsf{M}(n)$ base operations, and such that $\mathsf{M}(n)/n$ is non-decreasing (super-linearity hypothesis, see [GG03, p. 242]). For the particular case of p-adics with ground ring $R = \Bbbk[X]$, we can thus take $\mathsf{I}(n) = \mathsf{M}(n)$.

In the first subsection, we review known results for the function M, followed by algorithms for the short and middle product. We briefly mention the case $R = \mathbb{Z}$, and then the general case, at the end of this section.

1.2.1 Plain multiplication of polynomials

We recall here the three main multiplication algorithms: the naive, Karatsuba's and the FFT algorithm. Given $a, b \in \Bbbk[X]_{<n}$ of degree less than n, we want to compute the product $c = a\,b \in \Bbbk[X]$ of a and b.

The naive algorithm computes the n^2 terms

$$c = \sum_{0 \leqslant i,j < n} a_i\, b_j\, X^{i+j}.$$

Therefore this algorithm performs n^2 multiplications and $(n-1)^2$ additions in $\Bbbk$.

The first subquadratic algorithm for multiplication was given by Karatsuba [KO63]. This algorithm starts by splitting the polynomial inputs in two halves:

$$a_{0\ldots n} = a_{0\ldots m} + a_{m\ldots n}\, X^m, \quad b_{0\ldots n} = b_{0\ldots m} + b_{m\ldots n}\, X^m$$

where $m := \lfloor n/2 \rfloor$. Then we compute three half-sized multiplications

$$d := a_{0\ldots m}\, b_{0\ldots m}, \quad e := (a_{0\ldots m} + a_{m\ldots n})\,(b_{0\ldots m} + b_{m\ldots n}), \quad f := a_{m\ldots n}\, b_{m\ldots n}.$$

Finally we recombine linearly these products to get

$$c := d + (e - d - f)\, X^m + f\, X^{2m}.$$

Therefore if $\mathsf{K}(n)$ denotes the cost of Karatsuba's multiplication algorithm for polynomials of degree less than n, one has

$$\mathsf{K}(n) = \mathsf{K}(\lfloor n/2 \rfloor) + 2\,\mathsf{K}(\lceil n/2 \rceil) + \mathcal{O}(n)$$

leading to $\mathsf{K}(n) = \mathcal{O}(n^{\log_2(3)})$.

The principle of Karatsuba's algorithm is related to an evaluation/interpolation at points $0, 1$ and $+\infty$. More general evaluation/interpolation schemes can be found in the algorithms of Toom-Cook [Too63, Coo66]. For any $\alpha > 1$, there exists a Toom-Cook algorithm that runs in time $\mathcal{O}(n^\alpha)$.

The paper of Cooley and Tukey [CT65] founded the area of multiplication algorithms based on Fourier transforms. Let us describe the fast Fourier transform (FFT) algorithm, over a field $\Bbbk$. Let $m := 2^e$ be the smallest power of two greater or equal to $2\,n$. We start by assuming that there exists a mth primitive root of unity ω in $\Bbbk$. The discrete Fourier transform is the $\Bbbk$-linear isomorphism

$$\begin{array}{rccc} \mathrm{DFT}_\omega: & \Bbbk^m & \longrightarrow & \Bbbk^m \\ & (p_0, \ldots, p_{m-1}) & \longmapsto & (P(1), P(\omega), \ldots, P(\omega^{m-1})) \end{array}$$

where $P := \sum_{i=0}^{m-1} p_i\, X^i$. So DFT induces a bijection between $\Bbbk[X]_{<m}$ and $\Bbbk^m$. This transformation gives a new representation $(P(1), P(\omega), \ldots, P(\omega^{m-1}))$ of the polynomial P. The important fact is that the multiplication in $\Bbbk^m$ costs m multiplications, which is optimal.

Let us focus of the computation of DFT_ω and its inverse morphism DFT_ω^{-1}. A first important result is that

$$\mathrm{DFT}_\omega \circ \mathrm{DFT}_{\omega^{-1}} = \mathrm{DFT}_{\omega^{-1}} \circ \mathrm{DFT}_\omega = m\,\mathrm{Id}$$

and consequently

$$(\mathrm{DFT}_\omega)^{-1} = \frac{1}{m}\,\mathrm{DFT}_{\omega^{-1}}.$$

It remains to give a fast algorithm to compute the discrete Fourier transform. A divide-and-conquer strategy is used. Write $P = P_0 + P_1\, X^{m/2}$ and let $R_0 = P_0 + P_1 \in \Bbbk[X]_{m/2}$ and $R_1(X) = (P_0 - P_1)(\omega\, X) \in \Bbbk[X]_{m/2}$. Then for $0 \leqslant i < m/2$, one has

$$\begin{aligned} P(\omega^{2i}) &= P_0(\omega^{2i}) + P_1(\omega^{2i}) = R_0(\omega^{2i}) \\ P(\omega^{2i+1}) &= P_0(\omega^{2i+1}) - P_1(\omega^{2i+1}) = R_1(\omega^{2i}). \end{aligned}$$

Therefore the computation of $\mathrm{DFT}_\omega(P)$ reduces to two calls $\mathrm{DFT}_{\omega^2}(R_0)$ and $\mathrm{DFT}_{\omega^2}(R_1)$ and $\mathcal{O}(n)$ additions and multiplications. If $\mathsf{FFT}(m)$ is the cost of computing the discrete Fourier transform DFT_ω, then

$$\mathsf{FFT}(m) = 2\,\mathsf{FFT}(m/2) + \mathcal{O}(m)$$

which gives $\mathsf{FFT}(m) = \mathcal{O}(m \log(m))$. Finally the cost of multiplying two polynomials of degree less than n is $3\,\mathsf{FFT}(m) + \mathcal{O}(m) = \mathcal{O}(n \log(n))$.

When no roots of unity of sufficient order are available in the base field, we use the idea developed in [SS71] and [CK91]. This algorithm adds virtual roots of unity to our base field; it actually applies to any ring and multiplies polynomials of degrees less than n in time $\mathcal{O}(n \log(n) \log(\log(n)))$.

Let us now sum up all these algorithms.

Theorem 1.7. *One can take* $\mathsf{M}(n) \in \mathcal{O}(n \log(n) \log(\log(n)))$*, and thus* $\mathsf{I}(n) \in \mathcal{O}(n \log(n) \log(\log(n)))$ *when* $R = \Bbbk[X]$ *and* $p = X$*.*

1.2.2 Middle product of polynomials

The concept of middle product was introduced in [HQZ04]. That paper stresses the importance of this new operation in computer algebra and uses it to speed-up the division and square-root of power series.

Let $a, b \in \Bbbk[X]$ with $\lambda(b) = n$. Then, we have seen that the *middle product* $\mathrm{MP}(a, b)$ of a and b is defined as the part $c_{n-1\ldots 2n-1}$ of the product $c := a\, b$, so that $\deg(\mathrm{MP}(a, b)) \leqslant n - 1$. Naively, the middle product is computed via the full multiplication $c := (a\, b) \bmod p^{2n-1}$, which is done in time $2\,\mathsf{M}(n)$, but, as we will see, this not optimal. We denote by $\mathsf{MP}(n)$ the arithmetic complexity of the middle product of two polynomials a, b with $\lambda(b) \leqslant n$.

The middle product is closely related to the *transposed multiplication* [BLS03, HQZ04]. Thus, we will use the *Tellegen principle* that relates the complexities of a linear algorithm and its transposed algorithm. A linear algorithm can be formalized by linear straight-line programs (s.l.p.), which are s.l.p.'s with only linear operations. We refer to [BCS97, Chapter 13] for a precise exposition.

Theorem 1.8. ([BCS97, Th. 13.20]) *Let* $\Phi\colon R^n \to R^m$ *be a linear map that can be computed by a linear straight-line program of size* L *and whose matrix in the canonical bases has no zero rows or columns. Then the transposed map* Φ^t *can be computed by a linear straight-line program of size* $L - n + m$*.*

As it turns out, the middle product is a transposed multiplication, up to the reversal of polynomial. We deduce the following complexity result.

Corollary 1.9. *The complexity* MP *of the middle product satisfies*

$$\mathsf{MP}(n) = \mathsf{M}(n) + n - 1.$$

More precisely, while the number of additions can slightly differ between the multiplication and the middle product, the number of multiplications remains the same [HQZ04, Theorem 4].

Tellegen's principle gives more than the existence of a middle product algorithm with good complexity, it tells you how to build the transposed algorithm. It was first pointed out in [BLS03] that the transposition of algorithms can be done systematically and automatically (and the paper [DFS10] actually specifies an algorithm for automatic transposition based on the language *transalpyne*). We give a brief description of the middle product mechanisms corresponding to the transposition of the naive, Karatsuba and FFT multiplication algorithms.

Let us begin by a diagram. If we represent the polynomial coefficients $(a_i)_{0\leqslant i<2n-1}$ of a in abscissa and the coefficients $(b_j)_{0\leqslant j<n}$ of b in ordinate, the unit square whose left bottom corner is at coordinates $(i,\ j)$ corresponds to the elementary product $a_i\ b_j$. The big white square includes all the terms involved in the plain multiplication $a\ b$. The terms involved in the middle product $\mathrm{MP}(a,\ b)$ form a gray rhombus on the diagram.

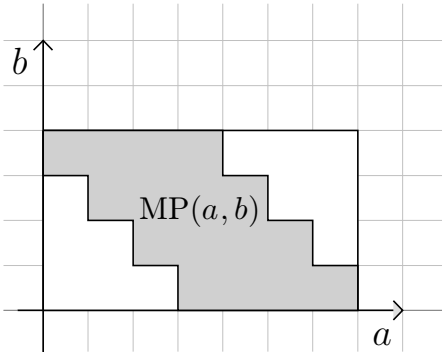

Figure 1.1. Plain and middle multiplication of polynomials

The naive multiplication algorithm gives the easiest scheme for middle product: only compute the coefficients $c_i = \sum_{j=0}^{n-1} a_{i-j}\, b_j$ of c for $n-1 \leqslant i < 2\,n-1$. This costs n^2 multiplications and $n\,(n-1)$ additions. Of course, the number of multiplications is the same as for the multiplication and the difference in the number of additions is predicted by Corollary 1.9. Indeed the difference of the number of additions between middle product and multiplication is $n\,(n-1) = (n-1)^2 + n - 1$.

Next, we sketch the Karatsuba middle product in the case of even length $n = \lambda(b)$.

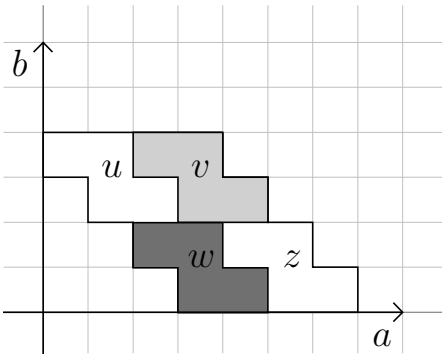

Figure 1.2. Karatsuba middle product on polynomials

The trick is to divide the diamond-shaped area of the middle product $\mathrm{MP}(a, b)$ into four parts u, v, w and z as depicted in Figure 1.2, that is

$$\begin{aligned} u &:= \mathrm{MP}(A_0, B_1) \\ v &:= \mathrm{MP}(A_1, B_1) \\ w &:= \mathrm{MP}(A_1, B_0) \\ z &:= \mathrm{MP}(A_2, B_0) \end{aligned}$$

where $A_0 := a_{0\ldots n-1}$, $A_1 := a_{n/2\ldots 3n/2-1}$, $A_2 := a_{n\ldots 2n-1}$, $B_0 := b_{0\ldots n/2}$ and $B_1 := b_{n/2\ldots n}$. Then we observe that by bilinearity $u + v = \mathrm{MP}(A_0 + A_1, B_1)$, $v - w = \mathrm{MP}(A_1, B_1 - B_0)$ and $w + z = \mathrm{MP}(A_1 + A_2, B_0)$. Therefore we get

$$\begin{aligned} \mathrm{MP}(a,b)_{0\ldots n/2} &= (u+v)-(v-w) \\ \mathrm{MP}(a,b)_{n/2\ldots n} &= (w+z)+(v-w). \end{aligned}$$

So we have reduced the problem of Karatsuba middle product to three half-sized recursive calls and a few additions. The case of odd length n is similar but somewhat more complicated. This algorithm is the transposed algorithm of Karatsuba's multiplication [HQZ04, BLS03].

For the FFT variant, and suppose that $\omega \in \Bbbk$ is a primitive $(2\,n-1)$th root of unity. We cut the product $c = a\,b$ in three parts $c_{0\ldots n-1}$, $c_{n-1\ldots 2n-1}$ and $c_{2n-1\ldots 3n-2}$ and remark that

$$(c_{0\ldots n-1} + c_{2n-1\ldots 3n-2}) + X^{n-1}\, c_{n-1\ldots 2n-1} = c \bmod (X^{2n-1} - 1). \tag{1.1}$$

Consequently given

$$\begin{aligned} a(1), \ldots, a(\omega^{2n-2}) &:= \mathrm{FFT}(a, \omega) \\ b(1), \ldots, b(\omega^{2n-2}) &:= \mathrm{FFT}(b, \omega) \end{aligned}$$

we reconstruct $e := c \bmod (X^{2n-1} - 1)$ by $e = \frac{1}{2\,n-1}\,\mathrm{FFT}\big(\big(\sum_{i=0}^{2n-2} a(\omega^i)\, b(\omega^i)\, X^i\big), \omega^{-1}\big)$. So finally $\mathrm{MP}(a,b) = e_{n-1\ldots 2n-1}$. In practice, we only work with 2^ℓth root of unity and a padding with zeroes is necessary to adjust Formula (1.1).

1.2.3 Short product of polynomials

We denote by $\Bbbk[X]_{<n}$ the set of polynomials of length lesser or equal to n. Let a, $b \in \Bbbk[X]_{<n}$ and define the *short product* $\mathrm{SP}(a,b)$ of a and b as the part $c_{0\ldots n}$ of the product $c := a\,b$. In other words, $c = (a\,b) \bmod X^n$. We denote by $\mathsf{SP}(n)$ the cost of the short product of $a, b \in \Bbbk[X]_{<n}$ and C_{SP} the ratio with plain multiplication, *i.e.* a constant such that $\mathsf{SP}(n) \leqslant C_{\mathsf{SP}}\,\mathsf{M}(n)$ for all $n \in \mathbb{N}^*$.

The situation with the short product is more contrasted than for the middle product. Although the size of the output is halved, we seldom gain a factor 2 in the cost: the actual cost of the short product is hard to pin down.

As always, it is easy to adapt the naive multiplication algorithm to compute only the first terms. In this case, we gain a factor two in the complexity, *i.e.* $C_{\mathsf{SP}} = 1/2$.

Let us now consider Karatsuba's multiplication. The paper [Mul00] published the first approach for having $C_{\mathsf{SP}} < 1$ for the cost function $\mathsf{M}(n) = n^{\log_2(3)}$, which is an approximation of the cost of Karatsuba's multiplication. The basic idea is to do two half-sized recursive calls and use a half-sized multiplication, but this does not improve the complexity. A refinement of previous idea consists in changing the size of the cutting of the problem and optimizing the complexity with respect to this size. It reaches a constant $C_{\mathsf{SP}} = 0.81$ for this approximated cost function. However, practical application of this method to Karatsuba's algorithm shows that the value 0.81 is not a upper bound of the ratio of timings but rather an estimation of its average.

The analysis of the ratio $\mathsf{SP}(n)/\mathsf{M}(n)$ for $\mathsf{M}(n)$ the exact number of multiplications of Karatsuba's algorithm is done [HZ04]. They find the optimal integer cutting for Karatsuba's short product and prove that $C_{\mathsf{SP}} = 1$ is the best upper bound.

However, the situation is different if we consider an hybrid multiplication algorithm that uses the naive, quadratic algorithm for small values and switches to Karatsuba's method for larger values. In this case, another variant based on odd/even decomposition [HZ04] performs well. This variant does three half-sized recursive calls and, intuitively, transfers the factor $C_{\mathsf{SP}} = 1/2$ attained by the naive methods to Karatsuba's. The paper show that, for a threshold $n_0 = 32$ between algorithm, one has $\mathsf{SP}(n) \leqslant 0.57\,\mathsf{M}(n)$ for $n > n_0$. The timings of the implementation of [HZ04] show that this factor $C_{\mathsf{SP}} = 0.6$ is also observed in practice in the degree range of the Karatsuba multiplication.

No improvement is known for the short product based on FFT multiplication. However, notice that we can compute $c_{0\ldots n} + c_{n\ldots 2n-1}$ in time $1/2\,\mathsf{M}(n)$ because it equals to c modulo $(X^n - 1)$. We will use this fact in Section 1.3.5 on middle relaxed multiplication.

1.2.4 The situation on integers

The presence of carries when computing with integers complicates the situation for all operations. Surprisingly, though, it is possible to obtain a slightly faster plain multiplication than in the polynomial case. We denote by $\log^*\colon \mathbb{R}_{>0} \to \mathbb{R}$ the iterated logarithm defined recursively by

$$\log^*(x) = \begin{cases} 0 & \text{if } 0 < x \leqslant 1 \\ 1 + \log^*(\log(x)) & \text{if } 1 < x \end{cases}.$$

Theorem 1.10. ([Für07, DKSS08]) *Two integers with n digits in base p can be multiplied in bit-complexity $\mathcal{O}(n \log(n)\, 2^{\log^*(n)})$.*

Note that this result involves a different complexity model than ours. It seems that the ideas of [DKSS08] could be adapted to give the same result in our complexity model.

As to middle and short product, few algorithms exist. Indeed, in the integer case, we face two kinds of problems, both due to carries. First, the middle and short product can have more than n coefficients. Moreover, the middle product $\mathrm{MP}(f, g)$ can no longer be seen as the middle part of their product.

As always, the naive algorithm adapts well for middle and short product of integers. The problems due to carries are solved for the Karatsuba middle product for integers in [Har12]. About the Karatsuba short product, we quote [HZ04]: "the carries are a simple matter to deal with in Mulders' method but are a real problem with our (odd/even) variant". Finally, it seems that the FFT middle product can be adapted to integers. Indeed if the middle product is not exactly the middle part of the multiplication, the difference concerns only a few of the lower and higher coefficients. Computing $a\ b$ modulo p^{2n-1}, we get most of the coefficients of the middle product and compute the missing coefficients in linear time.

We leave it as a future work to implement these methods and to assess their effect on the complexity of the relaxed multiplication of p-adic integers.

1.2.5 The situation on p-adics

Finally, we prove by a simple reduction that for any p-adic ring R_p, the cost function of off-line p-adic multiplication is always quasi-linear.

Theorem 1.11. *For any p-adic ring R_p, the cost $\mathsf{I}(n)$ of multiplication of p-adics of size n is bounded by $\mathcal{O}(\mathsf{M}(n)\log(n)^2)$.*

Proof. Let $a=\sum_{i=0}^{n-1} a_i\,p^i$ and $b=\sum_{i=0}^{n-1} b_i\,p^i$ be p-adics of length bounded by n. Introduce the polynomials $A=\sum_{i=0}^{n-1} a_i\,X^i$ and $B=\sum_{i=0}^{n-1} b_i\,X^i$ of $R[X]$ and let $C=\sum_{i=0}^{2n-2} c_i\,X^i\in R[X]$ be the product of A and B.

Since the length of the coefficients c_i of C is bounded by $\ell_c:=\lceil\log_2(n)\rceil+2$, we can multiply the polynomials A and B in the ring $(R/(p^{\ell_c}))[X]$ and recover C. Arithmetic operations in $(R/(p^{\ell_c}))$ can be computed in time $\mathcal{O}(\mathsf{I}(\log(n)))$, so we obtain C at cost $\mathcal{O}(\mathsf{I}(\log(n))\,\mathsf{M}(n))$; taking the naive bound $\mathsf{I}(n)=\mathcal{O}(n^2)$, we get the claimed cost $\mathcal{O}(\mathsf{M}(n)\log(n)^2)$.

Finally the p-adic $c:=a\,b$ equals to $C(p)$. In view of the third point in Lemma 1.1, the cost of the additions necessary to compute $C(p)$ is $\mathcal{O}(n\log(n))$. □

1.3 Relaxed algorithms for multiplication

In this section, we recall several relaxed algorithms for the on-line multiplication of p-adics, we analyze precisely their costs and give timings of our implementation using NTL. To our knowledge, no such precise comparison existed before.

Besides, we introduce a *new* relaxed multiplication algorithm using middle and short product, and show that it can perform better than some previous ones.

We start by recalling the state-of-the-art of on-line p-adic multiplication.

Theorem 1.12. ([FS74, Hoe97, BHL11]) *Whenever R_p is a power series ring or the ring of p-adic integers, the cost $\mathsf{R}(n)$ of multiplying two p-adics at precision n by an on-line algorithm is*

$$\mathcal{O}\left(\sum_{k=0}^{\lfloor\log_2(n)\rfloor}\frac{n}{2^k}\,\mathsf{I}(2^k)\right)=\begin{cases}\mathcal{O}(\mathsf{I}(n)) & \text{for naïve or Karatsuba's multiplication}\\ \mathcal{O}(\mathsf{I}(n)\log(n)) & \text{for FFT multiplication}\end{cases}.$$

The previous result was first discovered for integers in [FS74]; the details for the multiplication of power series were given in [Hoe97] and the paper [BHL11] generalizes relaxed algorithms for p-adic integers. The latter algorithm is correct for any p-adic ring but the authors analyze the complexity only for the p-adic integers. The issue with general p-adic rings is the management of carries. Although we do not prove it here, we believe that this complexity result carries forward to any p-adic ring.

Remark 1.13. Recent progress has been made on relaxed multiplication [Hoe07, Hoe12]. These papers give an on-line algorithm that multiplies power series on a wide range of rings, including all fields, in time

$$\mathsf{M}(n)\log(n)^{o(1)}.$$

Also, on-line multiplication of p-adic integers at precision n can be done in bit complexity

$$n \log(n)^{1+o(1)} \log(p) \log(\log(p)).$$

We will not give the details of these algorithms here.

In the next subsections, we will give a short presentation of the relaxed product algorithms that reach the bound of Theorem 1.12. Existing algorithms can be found in Sections 1.3.2, 1.3.3 and 1.3.4. Our new relaxed multiplication algorithm using short and middle product is presented in Section 1.3.5.

Although the algorithms are correct for any p-adic ring R_p, we will analyze their cost in the special case of power series rings: the exposition will be simplified since there are no carries.

To establish comparisons, and for the sake of completeness, we give for the first time *the constants hidden in the big-O notation* of the complexity estimates. All the following complexity analyses take into account only the number of basic multiplications, and do not count the basic additions. For the rest of this chapter, the *multiplicative complexity* of an algorithm is the number of basic multiplications it performs. We denote by M^* the multiplicative complexity function of polynomial multiplication. We sum up these bounds in the next two tables.

Table 1.1 gives bounds on the multiplicative complexity of *semi-relaxed* multiplication algorithms depending on the algorithm we use to multiply truncated power series (naive, Karatsuba or FFT).

The semi-relaxed multiplication algorithm which appears in [Hoe07] gives the costs of the first line; we give an overview of this algorithm in Section 1.3.2. The second line corresponds to the semi-relaxed algorithm using middle product presented in [Hoe03], which can be found in Section 1.3.3.

	naive	Karatsuba	FFT
semi-relaxed	$\leqslant 2\,\mathsf{M}^*(n)$	$\leqslant 3\,\mathsf{M}^*(n)$	$\sim\frac{1}{2}\,\mathsf{M}^*(n)\log_2(n)$
semi-relaxed with middle	$\leqslant 1.5\,\mathsf{M}^*(n)$	$\leqslant 2\,\mathsf{M}^*(n)$	$\sim\frac{1}{4}\,\mathsf{M}^*(n)\log_2(n)$

Table 1.1. Multiplicative complexity of the semi-relaxed multiplication of power series

Table 1.2 describes relaxed algorithms. The first line of Table 1.2 corresponds to the relaxed multiplication algorithm of [FS74, Hoe97, BHL11]. This algorithm is presented in Section 1.3.4. Our contribution, the relaxed multiplication using middle and short product, gives the results of the second line. It is presented in Section 1.3.5.

	naive	Karatsuba	FFT
relaxed	$\leqslant\mathsf{M}^*(n+1)$	$\leqslant 2.5\,\mathsf{M}^*(n+1)$	$\sim\mathsf{M}^*(n)\log_2(n)$
relaxed with short and middle	$\leqslant\mathsf{M}^*(n+1)$	$\leqslant\begin{cases} 1.75\,\mathsf{M}^*(n+1) & \text{if } C_{\mathsf{SP}}=\frac{1}{2} \\ 2.5\,\mathsf{M}^*(n+1) & \text{if } C_{\mathsf{SP}}=1 \end{cases}$	$\sim\frac{1}{2}\,\mathsf{M}^*(n)\log_2(n)$

Table 1.2. Multiplicative complexity of the relaxed multiplication of power series

Remark 1.14. It was remarked in [Hoe97, Hoe02] that the Karatsuba multiplication could be rewritten as a relaxed algorithm, thus leading to a relaxed multiplication algorithm with exactly the same numbers of operations.

However, this algorithm is often not practical. The rewriting induces $\Omega(\log(n))$ function calls at each step of the multiplication, which makes it very poorly suited to practical implementations. For these reasons, we will not study this algorithm.

Remark 1.15. When the required precision n is known in advance, it is possible to adapt the on-line multiplication algorithms to this specific precision and thus lower the bounds given in Tables 1.1 and 1.2 (see [Hoe02, Hoe03]). An example of such an algorithm is given in the paragraph "Link between divide-and-conquer and semi-relaxed" in Section 1.3.3. However, these considerations are not developed further in this thesis.

We choose to present a simple form of the relaxed product algorithms, which will be convenient to understand the operations made in the computation of recursive p-adics in the next chapter. We allocate ourselves the memory to store the current state of the computation and we indicate to the program at which step we are. If one were to implement these algorithms, our description would not be appropriate. We would recommend the implementation described in [Hoe02, BHL11], which is actually very close to the implementation in MATHEMAGIX [HLM+02].

1.3.1 Complexity preliminaries

We introduce three auxiliary complexity functions from $\mathbb{N}$ to $\mathbb{N}$,

$$\begin{aligned}
\mathsf{M}^{(1)}(n) &:= \sum_{k=0}^{\lfloor \log_2(n) \rfloor} \mathsf{M}^*(2^k) \\
\mathsf{M}^{(2)}(n) &:= \sum_{k=0}^{\lfloor \log_2(n) \rfloor} \left\lfloor \frac{n}{2^k} \right\rfloor \mathsf{M}^*(2^k) \\
\mathsf{M}^{(3)}(n) &:= \sum_{k=0}^{\lfloor \log_2(n) \rfloor} \left\lfloor \frac{n}{2^{(k+1)}} + \frac{1}{2} \right\rfloor \mathsf{M}^*(2^k).
\end{aligned}$$

These functions will be used afterwards to assess the multiplicative complexity of our (semi-)relaxed multiplication algorithms.

Lemma 1.16. *Let $\ell := \lfloor \log_2(n) \rfloor$ and $n = \sum_{i=0}^{\ell} \bar{n_i}\, 2^i$ be the base-2 expansion of n. Assume that $\mathsf{M}^*(1) = 1$ and that there exists $\alpha \in]1; +\infty[$ such that for all $n \in \mathbb{N}$, $\mathsf{M}^*(2\,n) = 2^\alpha\, \mathsf{M}^*(n)$. Then*

$$\begin{aligned}
\mathsf{M}^{(1)}(n) &:= \frac{2^\alpha}{2^\alpha - 1}\, \mathsf{M}^*(2^\ell) - \frac{1}{2^\alpha - 1} \\
\mathsf{M}^{(2)}(n) &:= \frac{2^\alpha}{2^\alpha - 2} \sum_{i=0}^{\ell} \bar{n_i}\, \mathsf{M}^*(2^i) - \frac{2\,n}{2^\alpha - 2} \\
\mathsf{M}^{(3)}(n) &:= \frac{2^\alpha - 1}{2^\alpha - 2} \sum_{i=0}^{\ell} \bar{n_i}\, \mathsf{M}^*(2^i) - \frac{n}{2^\alpha - 2}.
\end{aligned}$$

Proof. First

$$\mathsf{M}^{(1)}(n) = \sum_{k=0}^{\ell} 2^{\alpha k} = \frac{2^{\alpha(\ell+1)}-1}{2^{\alpha}-1} = \frac{2^{\alpha}}{2^{\alpha}-1}\,\mathsf{M}^*(2^{\ell}) - \frac{1}{2^{\alpha}-1}.$$

Next, one has

$$\begin{aligned}
\mathsf{M}^{(2)}(n) &= \sum_{k=0}^{\ell}\sum_{i=k}^{\ell} \bar{n_i}\, 2^{i-k}\,\mathsf{M}^*(2^k) \\
&= \sum_{i=0}^{\ell} \bar{n_i}\, 2^{i} \sum_{k=0}^{i} 2^{(\alpha-1)k} \\
&= \sum_{i=0}^{\ell} \bar{n_i}\, 2^{i}\, \frac{2^{(\alpha-1)(i+1)}-1}{2^{(\alpha-1)}-1} \\
&= \frac{2^{\alpha}}{2^{\alpha}-2} \sum_{i=0}^{\ell} \bar{n_i}\,\mathsf{M}^*(2^i) - \frac{2\,n}{2^{\alpha}-2}.
\end{aligned}$$

Finally, we have the equalities

$$\begin{aligned}
\mathsf{M}^{(3)}(n) &= \sum_{k=0}^{\ell} \left(\left\lfloor \frac{n}{2^{(k+1)}} \right\rfloor + \bar{n_k} \right) \mathsf{M}^*(2^k) \\
&= \sum_{k=0}^{\ell} \sum_{i=k+1}^{\ell} \bar{n_i}\, 2^{i-(k+1)}\,\mathsf{M}^*(2^k) + \sum_{k=0}^{\ell} \bar{n_k}\,\mathsf{M}^*(2^k) \\
&= \sum_{i=1}^{\ell} \bar{n_i}\, 2^{i-1} \sum_{k=0}^{i-1} 2^{(\alpha-1)k} + \sum_{k=0}^{\ell} \bar{n_k}\,\mathsf{M}^*(2^k) \\
&= \sum_{i=1}^{\ell} \bar{n_i}\, 2^{i-1}\, \frac{2^{(\alpha-1)i}-1}{2^{(\alpha-1)}-1} + \sum_{k=0}^{\ell} \bar{n_k}\,\mathsf{M}^*(2^k) \\
&= \left(\frac{1}{2^{\alpha}-2} \sum_{i=0}^{\ell} \bar{n_i}\,\mathsf{M}^*(2^i) - \frac{n}{2^{\alpha}-2} \right) + \sum_{k=0}^{\ell} \bar{n_k}\,\mathsf{M}^*(2^k) \\
&= \frac{2^{\alpha}-1}{2^{\alpha}-2} \sum_{i=0}^{\ell} \bar{n_i}\,\mathsf{M}^*(2^i) - \frac{n}{2^{\alpha}-2}.
\end{aligned}$$

□

Lemma 1.17. *Assume that* M^* *counts the number of multiplication of the naive or Karatsuba's algorithm. Let* $n = \sum_{i=0}^{\ell} \bar{n_i}\, 2^i$ *be the base-2 expansion of* n*. Then*

$$\mathsf{M}^*(2^{\ell}) + (\mathsf{M}^*(3) - \mathsf{M}^*(2)) \sum_{i=0}^{\ell-1} \bar{n_i}\,\mathsf{M}^*(2^i) \leqslant \mathsf{M}^*(n).$$

Proof. Under the same hypothesis on M^*, we start by proving that for any $n \geqslant 1$,

$$\mathsf{M}^*(2\,n) + (\mathsf{M}^*(3) - \mathsf{M}^*(2))\,\mathsf{M}^*(1) \leqslant \mathsf{M}^*(2\,n+1). \tag{1.2}$$

Let $C := (\mathsf{M}^*(3) - \mathsf{M}^*(2))$. For the naive multiplication algorithm, one has

$$C = 5 \leqslant \mathsf{M}^*(2\,n+1) - \mathsf{M}^*(2\,n) = 4\,n+1.$$

For Karatsuba's multiplication algorithm, we proceed as follows. Recall that the Karatsuba's cost function satisfies $\mathsf{M}^*(n) = 2\,\mathsf{M}^*(\lceil n/2 \rceil) + \mathsf{M}^*(\lfloor n/2 \rfloor)$. We have to prove the inequality

$$C = 4 \leqslant \mathsf{M}^*(2\,n+1) - \mathsf{M}^*(2\,n) = 2\,(\mathsf{M}^*(n+1) - \mathsf{M}^*(n)).$$

So we prove that for any $n \geqslant 1$, $\mathsf{M}^*(n+1) - \mathsf{M}^*(n) \geqslant 2$. First, $\mathsf{M}^*(2) - \mathsf{M}^*(1) = 2$ and $\mathsf{M}^*(3) - \mathsf{M}^*(2) \geqslant 2$. Then recursively, we assume that the result is true until $n \geqslant 2$ and prove it for $n+1$. We separate the odd and even cases. If $n+1 = 2\,k$, then $k \geqslant 1$ and

$$\mathsf{M}^*(n+2) - \mathsf{M}^*(n+1) = 2\,\mathsf{M}^*(k+1) + \mathsf{M}^*(k) - 3\,\mathsf{M}^*(k) = 2\,(\mathsf{M}^*(k+1) - \mathsf{M}^*(k)) \geqslant 4.$$

Else, if $n+1 = 2\,k+1$, then $k \geqslant 1$ and

$$\mathsf{M}^*(n+2) - \mathsf{M}^*(n+1) = 3\,\mathsf{M}^*(k+1) - (2\,\mathsf{M}^*(k+1) + \mathsf{M}^*(k)) = \mathsf{M}^*(k+1) - \mathsf{M}^*(k) \geqslant 2.$$

So Equation (1.2) is proved and we can prove the lemma. First,

$$\begin{aligned} \mathsf{M}^*(2^\ell) + C\,\bar{n}_{\ell-1}\,\mathsf{M}^*(2^{\ell-1}) &= 3^{\ell-1}\,(\mathsf{M}^*(2) + C\,\bar{n}_{\ell-1}\,\mathsf{M}^*(1)) \\ &\leqslant 3^{\ell-1}\,\mathsf{M}^*(2 + \bar{n}_{\ell-1}\cdot 1) \\ &= \mathsf{M}^*(2^\ell + \bar{n}_{\ell-1}\cdot 2^{\ell-1}). \end{aligned}$$

Then

$$\begin{aligned} \mathsf{M}^*(2^\ell) + C\sum_{i=\ell-2}^{\ell-1} \bar{n_i}\,\mathsf{M}^*(2^i) &\leqslant \mathsf{M}^*(2^\ell + \bar{n}_{\ell-1}\cdot 2^{\ell-1}) + C\,\bar{n}_{\ell-2}\,\mathsf{M}^*(2^{\ell-2}) \\ &= 3^{\ell-2}\,(\mathsf{M}^*(4 + \bar{n}_{\ell-1}\cdot 2) + C\,\bar{n}_{\ell-2}\,\mathsf{M}^*(1)) \\ &\leqslant 3^{\ell-2}\,\mathsf{M}^*(4 + \bar{n}_{\ell-1}\cdot 2 + \bar{n}_{\ell-2}\cdot 1) \\ &= \mathsf{M}^*(2^\ell + \bar{n}_{\ell-1}\cdot 2^{\ell-1} + \bar{n}_{\ell-2}\cdot 2^{\ell-2}). \end{aligned}$$

We repeat this process until we have

$$\mathsf{M}^*(2^\ell) + C\sum_{i=0}^{\ell-1} \bar{n_i}\,\mathsf{M}^*(2^i) \leqslant \mathsf{M}^*\left(\sum_{i=0}^{\ell} \bar{n_i}\,2^i\right) = \mathsf{M}^*(n). \qquad \square$$

Lemma 1.18. *If* $\mathsf{M}^*(n) = K\,n\,(\log_2 n)^i\,(\log_2\log_2 n)^j$ *with* $K \in \mathbb{R}_{>0}, (i,j) \in \mathbb{N}^2$, *one has*

$$\begin{aligned} \mathsf{M}^{(2)}(n) &\sim_{n\to\infty} \frac{1}{(i+1)}\,\mathsf{M}^*(n)\log_2(n) \\ \mathsf{M}^{(3)}(n) &\sim_{n\to\infty} \frac{1}{2\,(i+1)}\,\mathsf{M}^*(n)\log_2(n). \end{aligned}$$

Proof. We set the notation $\ell := \lfloor \log_2(n) \rfloor$. As M^* is a super-linear function, we get

$$\mathsf{M}^{(1)}(n) \;\leqslant\; \sum_{k=0}^{\ell} \frac{1}{2^{\ell-k}}\, \mathsf{M}^*((2^\ell/2^k)\, 2^k) \leqslant 2\, \mathsf{M}^*(n).$$

Also, one has

$$\mathsf{M}^{(2)}(n) = \sum_{k=0}^{\ell} \lfloor n/2^k \rfloor\, \mathsf{M}^*(2^k) = \sum_{k=0}^{\ell} (n/2^k)\, \mathsf{M}^*(2^k) + \mathcal{O}_{n\to\infty}\Big(\mathsf{M}^{(1)}(n)\Big).$$

Since $\mathsf{M}^{(1)}(n) = \mathcal{O}_{n\to\infty}(\mathsf{M}^*(n))$ and since one has for $n \to \infty$

$$\begin{aligned}
\sum_{k=0}^{\ell} (n/2^k)\, \mathsf{M}^*(2^k) \;&\sim\; K \sum_{k=0}^{\ell} (n/2^k)\, 2^k\, k^i \log_2^j(k) \\
&\sim\; K\, n \left(\sum_{k=0}^{\ell} k^i \log_2^j(k) \right) \\
&\sim\; K\, n \left(\frac{\ell^{i+1}}{i+1} \log_2^j(\ell) \right)
\end{aligned}$$

we deduce that $\mathsf{M}^{(2)}(n) \sim_{n\to\infty} \frac{1}{(i+1)}\, \mathsf{M}^*(n) \log_2(n)$. Finally, we deal with $\mathsf{M}^{(3)}$:

$$\mathsf{M}^{(3)}(n) = \sum_{k=0}^{\ell} \frac{n}{2^{k+1}}\, \mathsf{M}^*(2^k) + \mathcal{O}_{n\to\infty}\Big(\mathsf{M}^{(1)}(n)\Big) \sim_{n\to\infty} \frac{1}{2\,(i+1)}\, \mathsf{M}^*(n) \log_2(n). \qquad \square$$

1.3.2 Semi-relaxed multiplication

The forthcoming half-line algorithm for the multiplication of p-adics was introduced in [FS74, Hoe03]. We briefly recall its mechanism. To do the product of p-adics a and b, we use extra inputs $c \in R_p$ and $i \in \mathbb{N}$: the p-adic c stores the current state of the computation and the integer i indicates at which step we are. The `SemiRelaxedProductStep` algorithm requires multiplications between finite precision p-adics. Because the required coefficients of a and b are known at that moment, any multiplication algorithm that takes as input truncated p-adics can be used.

We denote by $\nu_2(n)$ the valuation in 2 of the integer n. We obtain the following algorithm of which a is the only on-line argument.

Algorithm `SemiRelaxedProductStep`
Input: $a, b, c \in R_p$ and $i \in \mathbb{N}$ **Output:** $c \in R_p$
1. **for** k from 0 to $\nu_2(i+1)$
a. $c = c + a_{i-2^k+1\ldots i+1}\, b_{2^k-1\ldots 2^{k+1}-1}\, p^i$
2. **return** c

The diagram in Figure 1.3 will help us to understand the multiplications done in Algorithm `SemiRelaxedProductStep`. The coefficients $a_0, a_1, \ldots$ of a are placed in abscissa and the coefficients $b_0, b_1, \ldots$ of b in ordinate. Each unit square corresponds to a product between corresponding coefficients of a and b, *i.e.* the unit square whose left-bottom corner is at coordinates (i, j) stands for $a_i\, b_j$. Each bigger square corresponds to a product of finite precision p-adics; an $s \times s$ square whose left-bottom corner is at coordinates (i, j) stands for $a_{i\ldots i+s}\, b_{j\ldots j+s}$. The number inside the square indicates at which step this computation is done.

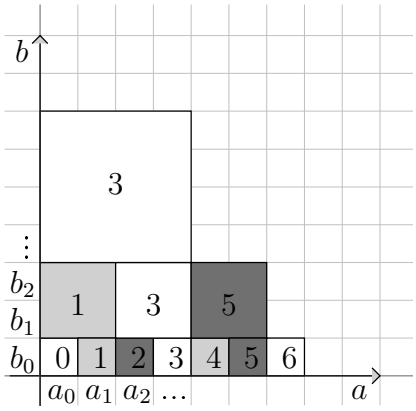

Figure 1.3. Semi-relaxed multiplication

We define two properties for any algorithm `Algo` with entries in $R_p^3 \times \mathbb{N}$ and output in R_p. These properties check that the algorithm computes progressively the product of the first two entries. The property $(\mathcal{HL})$ is the half-line variant and the property $(\mathcal{OL})$ is the on-line variant.

Property $(\mathcal{HL})$: For any $n \in \mathbb{N}$ and any a, b, $c_0 \in R_p$, the result $c \in R_p$ of the computation

Algorithm $\texttt{Loop}_{\texttt{Algo}}$
Input: $a, b, c_0 \in R_p$ and $n \in \mathbb{N}$ **Output:** $c \in R_p$
1. $c = c_0$
2. **for** i from 0 to n
a. $c = \texttt{Algo}(a, b, c, i)$
3. **return** c

satisfies $c = c_0 + a\, b$ modulo p^{n+1}. Moreover, during the computation, the Turing machine reads at most the coefficients $a_0, \ldots, a_n$ of the input a.

Property $(\mathcal{OL})$: Algorithm `Algo` must satisfy Property $(\mathcal{HL})$ and, additionally, read at most the coefficients $b_0, \ldots, b_n$ of the input b.

Property $(\mathcal{HL})$ states that the algorithm `Algo` is half-line and increments the number of correct p-adic coefficients of the product $c = a\,b$. This is the case for our algorithm `SemiRelaxedProductStep`.

Proposition 1.19. *Algorithm* `SemiRelaxedProductStep` *satisfies Property* $(\mathcal{HL})$.

We can check on Figure 1.3 that for all $n \in \mathbb{N}$, all the coefficients of the product $a\,b = \sum_{i=0}^{n} \sum_{j=0}^{i} a_j\,b_{i-j}\,p^i$ modulo p^{n+1} are computed by the semi-relaxed product before or at step n. We can also check that the algorithm is half-line in a since at step i, we use at most the coefficients $a_0, ..., a_i$ of a. However the operand b is off-line because, for example, it reads the coefficients $b_0, ..., b_6$ of b at step 3.

Complexity analysis As said before, we analyze the cost in the special case of R_p being a power series ring. For this reason, truncated p-adics are polynomials and their multiplication cost is denoted by $\mathsf{M}^*(n)$. For the sake of clarity, Algorithm $\mathtt{Loop}_{\mathtt{SemiRelaxedProductStep}}$ will also be called Algorithm `SemiRelaxedProduct`.

The cost $\mathsf{SR}^*(n)$ of all the off-line polynomial multiplications in the semi-relaxed algorithm `SemiRelaxedProduct` up to precision n (*i.e.* the terms in p^i for $0 \leqslant i < n$) is exactly $\mathsf{M}^{(2)}(n)$. Indeed, we do at each step a product of polynomials of degree 0 which each costs $\mathsf{M}^*(1) = 1$. We do every other step, starting for step 1, a product of polynomials of degree 1 which each costs $\mathsf{M}^*(2)$ and so on.

Proposition 1.20. *One has*

$$\mathsf{SR}^*(n) \leqslant \begin{cases} 2\,\mathsf{M}^*(n) & \text{for the naive multiplication} \\ 3\,\mathsf{M}^*(n) & \text{for Karatsuba's multiplication} \end{cases}$$

and these bound are asymptotically optimal since

$$\mathsf{SR}^*(2^m) \sim_{m\to\infty} \begin{cases} 2\,\mathsf{M}^*(2^m) & \text{for the naive multiplication} \\ 3\,\mathsf{M}^*(2^m) & \text{for Karatsuba's multiplication.} \end{cases}$$

Moreover when $\mathsf{M}^*(n) = K\,n\log_2(n)\log_2(\log_2(n))$ *with* $K \in \mathbb{R}_{>0}$, *one has*

$$\mathsf{SR}^*(n) \sim_{n\to\infty} \frac{1}{2}\,\mathsf{M}^*(n)\log_2(n).$$

Proof. Let us begin with the case where M^* is the cost function of the naive or Karatsuba's multiplication. Using Lemma 1.16 for the first equality and Lemma 1.17 for the second inequality, we have that for all $n \in \mathbb{N}$,

$$\begin{aligned} \mathsf{SR}^*(n) &= \frac{2^\alpha}{2^\alpha - 2} \sum_{i=0}^{\ell} \bar{n_i}\,\mathsf{M}^*(2^i) - \frac{2\,n}{2^\alpha - 2} \\ &\leqslant \frac{2^\alpha}{2^\alpha - 2}\,\mathsf{M}^*(n) + 0. \end{aligned}$$

When $n = 2^m$, one has

$$\mathsf{SR}^*(2^m) = \frac{2^\alpha}{2^\alpha - 2}\,\mathsf{M}^*(2^m) - \frac{2\cdot 2^m}{2^\alpha - 2} \sim_{m\to\infty} \frac{2^\alpha}{2^\alpha - 2}\,\mathsf{M}^*(2^m).$$

At last, when $\mathsf{M}^*(n) = K\, n \log_2(n) \log_2(\log_2(n))$, we use Lemma 1.18 to obtain

$$\mathsf{SR}^*(n) \sim_{n\to\infty} \frac{1}{2}\,\mathsf{M}^*(n) \log_2(n). \qquad \square$$

This gives the entries of the first line in Table 1.1, keeping in mind that the cost of additions $\mathcal{O}(n \log(n))$ is omitted.

1.3.3 Semi-relaxed multiplication with middle product

Another semi-relaxed algorithm, using middle products, was introduced in [Hoe03]. Whereas the semi-relaxed product `SemiRelaxedProduct` used plain multiplication on truncated p-adics as a basic tool, middle products are used to compute incrementally the product $a\,b$. Naturally, the following algorithm is of interest when there exists efficient middle and short product algorithms, e.g. when $R_p = \Bbbk[[X]]$. This algorithm is on-line with respect to the input a.

Algorithm `SemiRelaxedProductMiddleStep`
Input: $a, b, c \in R_p$ and $i \in \mathbb{N}$ **Output:** $c \in R_p$
1. Let $m := \nu_2(i+1)$
2. $c = c + \mathrm{MP}(a_{i-2^m+1\ldots i+1}, b_{0\ldots 2^{m+1}-1})\, p^i$
3. **return** c

The mechanism of the algorithm is sketched in Figure 1.4.

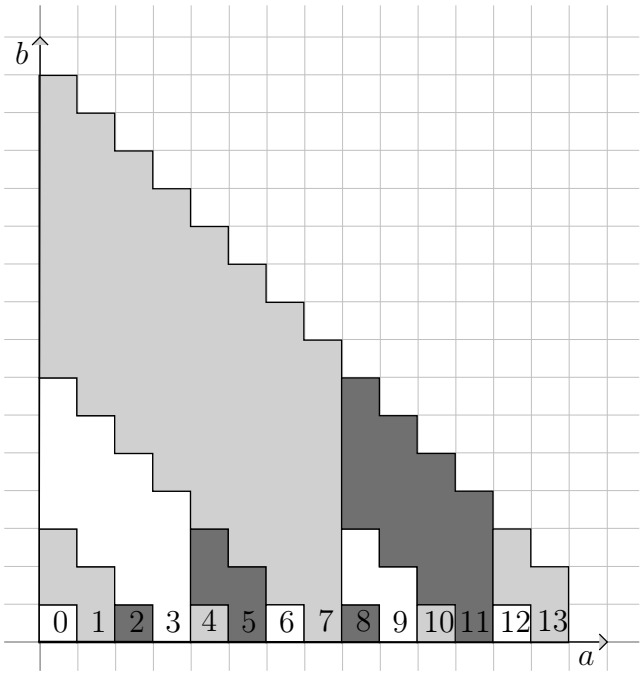

Figure 1.4. Semi-relaxed multiplication with middle product

Proposition 1.21. *Algorithm* `SemiRelaxedProductMiddleStep` *satisfies Property* $(\mathcal{HL})$.

This algorithm is still half-line for a because at step i, only the coefficients $a_0, \ldots,$ a_i are required. The input argument b is off-line because, for example, at step 3 the algorithms reads $b_0, \ldots, b_6$.

Complexity analysis Let MP^* be the multiplicative complexity function of the middle product. The multiplicative complexity $\mathsf{SRM}^*(n)$ of the semi-relaxed multiplication algorithm `SemiRelaxedProductMiddle`, that is Algorithm $\mathtt{Loop}_{\mathtt{SemiRelaxedProductMiddleStep}}$, for power series up to precision n is

$$\mathsf{SRM}^*(n) = \sum_{k=0}^{\lfloor \log_2(n) \rfloor} \left\lfloor \frac{(n+2^k)}{2^{k+1}} \right\rfloor \mathsf{MP}^*(2^k).$$

Indeed, as we can see on Figure 1.4, we do a middle product of degree 2^k each 2^{k+1} step starting from step $2^k - 1$.

Proposition 1.22. *One has*

$$\mathsf{SRM}^*(n) \leqslant \begin{cases} 1.5\,\mathsf{M}^*(n) & \text{for the naive multiplication} \\ 2\,\mathsf{M}^*(n) & \text{for Karatsuba's multiplication} \end{cases}$$

and these bound are asymptotically optimal since

$$\mathsf{SRM}^*(2^m) \sim_{m\to\infty} \begin{cases} 1.5\,\mathsf{M}^*(2^m) & \text{for the naive multiplication} \\ 2\,\mathsf{M}^*(2^m) & \text{for Karatsuba's multiplication.} \end{cases}$$

Moreover when $\mathsf{M}^*(n) = K\, n \log_2(n) \log_2(\log_2(n))$ *with* $K \in \mathbb{R}_{>0}$, *one has*

$$\mathsf{SRM}^*(n) \sim_{n\to\infty} \frac{1}{4}\,\mathsf{M}^*(n) \log_2(n).$$

This proposition gives the entries of the second line in Table 1.1.

Proof. Let $\ell := \lfloor \log_2(n) \rfloor$. Since $\mathsf{MP}^*(n) = \mathsf{M}^*(n)$ (see Section 1.2.2), we deduce that

$$\mathsf{SRM}^*(n) := \sum_{k=0}^{\lfloor \log_2(n) \rfloor} \left\lfloor \frac{(n+2^k)}{2^{k+1}} \right\rfloor \mathsf{M}^*(2^k) = \mathsf{M}^{(3)}(n).$$

We start by taking M^* the cost function of the naive or Karatsuba's multiplication. By Lemma 1.16 and Lemma 1.17, one has

$$\mathsf{SRM}^*(n) = \mathsf{M}^{(3)}(n) = \frac{2^\alpha - 1}{2^\alpha - 2} \sum_{i=0}^{\ell} \bar{n_i}\, \mathsf{M}^*(2^i) - \frac{n}{2^\alpha - 2} \leqslant \frac{2^\alpha - 1}{2^\alpha - 2}\, \mathsf{M}^*(n).$$

When $n = 2^m$, one has

$$\mathsf{SRM}^*(2^m) = \frac{2^\alpha - 1}{2^\alpha - 2}\, \mathsf{M}^*(2^m) - \frac{2^m}{2^\alpha - 2} \sim_{m\to\infty} \frac{2^\alpha - 1}{2^\alpha - 2}\, \mathsf{M}^*(2^m).$$

Finally in the case where $\mathsf{M}^*(n) = K\, n \log_2(n) \log_2(\log_2(n))$ with $K \in \mathbb{R}_{>0}$, we use Lemma 1.18 to get

$$\mathsf{SRM}^*(n) \sim_{n\to\infty} \frac{1}{4}\mathsf{M}^*(n)\log_2(n). \qquad \square$$

Link between divide-and-conquer and semi-relaxed In fact, the algorithm of [Hoe03], referred as the DAC algorithm from now on, is a little bit different. It is based on the following divide-and-conquer approach. Let us fix the desired precision n in advance. The computation of $c = a\,b$ at precision n reduces to the computation of $c_{0\ldots k}$, $\mathrm{MP}(a_{0\ldots\ell}, b_{n+1-2\ell\ldots n})$ and $d_{0\ldots k}$ where $k := \lfloor n/2 \rfloor$, $\ell := \lceil n/2 \rceil$ and $d := a_{\ell\ldots n} b_{0\ldots k}$. Then

$$c_{0\ldots n} = c_{0\ldots k} + \mathrm{MP}(a_{0\ldots\ell}, b_{n+1-2\ell\ldots n})\, p^{n+1-\ell} + d_{0\ldots k}\, p^{\ell}.$$

This cutting of the problem can be seen geometrically on Figure 1.5.

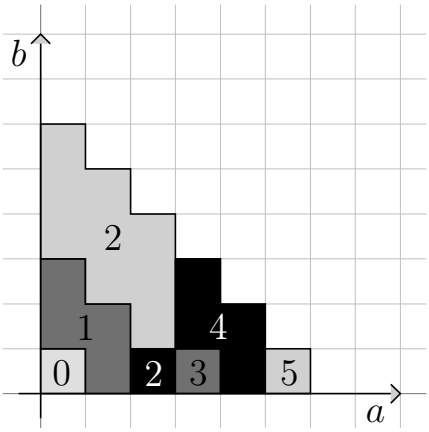

Figure 1.5. Divide-and-conquer truncated p-adics multiplication for $n = 6$

We have to compute the terms of the product inside a triangle. We make the biggest rhombus fit in the top left corner of the triangle; this corresponds to the middle product we do. Then the remaining area is the union of two triangles, which corresponds to two recursive calls. Now the DAC algorithm just reorders the computation so it can be relaxed up to precision n. Notice that our algorithm coincides with the DAC algorithm for precisions n that are powers of two minus one.

The first difference with our algorithm is that the scheme of computation of the DAC algorithm is adapted to the precision n; at step $n-1$, no unnecessary term of the product has been computed in the DAC algorithm. This differs with our algorithm which of course anticipates some computations. Therefore the DAC algorithm compares better to the off-line multiplication algorithm of Section 1.2.1.

Because the semi-relaxed multiplication using middle product comes from a divide-and-conquer approach, we should not be surprised if some relaxed algorithms for further problems based on this implementation of the multiplication coincides with divide-and-conquer algorithms. We will encounter two examples during this thesis: when solving a linear system over p-adics in Chapter 3 and when solving singular linear differential equations in Chapter 4.

1.3.4 Relaxed multiplication

Historically, the computation scheme of the forthcoming algorithm came from the on-line multiplication for integers of [FS74]. Then came the on-line multiplication for real numbers in [Sch97], and relaxed multiplication for power series [Hoe97, Hoe02], improved in [Hoe07] for some ground fields. This algorithm was extended to the multiplication of p-adic integers in [BHL11]. It is on-line with respect to both inputs a and b.

Algorithm `RelaxedProductStep`

Input: $a, b, c \in R_p$ and $i \in \mathbb{N}$
Output: $c \in R_p$

1. **for** k from 0 to $\nu_2(i+2)$
   a. $c = c + a_{i+1-2^k \dots i+1}\, b_{2^k-1 \dots 2^{k+1}-1}\, p^i$
   b. **if** $(i+2 = 2^{k+1})$
      return c
   c. $c = c + a_{2^k-1 \dots 2^{k+1}-1}\, b_{i+1-2^k \dots i+1}\, p^i$
2. **return** c

Here is a diagram that sums up the computation made at each step. We can see on this figure that the algorithm is online and that at step i, the product is correct up to at least precision $i+1$.

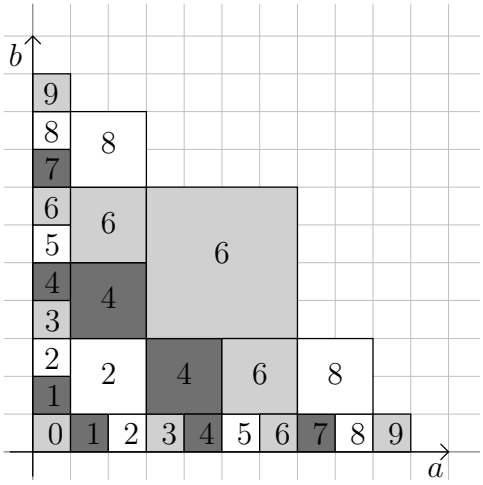

Figure 1.6. Relaxed multiplication

The relaxed algorithm is built recursively with the help of the semi-relaxed product. Suppose the relaxed product algorithm is constructed up to precision $2^m - 1$. Then one can extend it up to precision $2^{m+1} - 1$ with two semi-relaxed algorithms for $a_{2^m-1\dots\infty}\, b$ and $a\, b_{2^m-1\dots\infty}$. Then at precision $2^{m+1} - 1$, one completes the computations with the product $a_{2^m-1\dots 2^{m+1}-1}\, b_{2^m-1\dots 2^{m+1}-1}$ to obtain the terms $\sum_{0 \leqslant i,j \leqslant 2^{m+1}-1} a_i\, b_j\, p^{i+j}$ of the product $a\, b$. This construction is more obvious in Figure 1.6, where we identify the diagrams of the two semi-relaxed products.

Proposition 1.23. *Algorithm* `RelaxedProductStep` *satisfies Property* $(\mathcal{OL})$.

Once again, Figure 1.6 is of great help to see that the relaxed product algorithm does indeed compute the product $a\,b$. It is also easy to check that the algorithm is on-line on the diagram.

Complexity analysis Denote by $\mathsf{R}^*(n)$ the cost induced by all off-line multiplications done up to precision n, in the case where $R=\Bbbk[X]$. We can express it as

$$\mathsf{R}^*(n)=\sum_{k=0}^{\lfloor \log_2(n+1)\rfloor-1}\left(2\left\lfloor\frac{n+1}{2^k}\right\rfloor-3\right)\mathsf{M}^*(2^k).$$

Proposition 1.24. *One has*

$$\mathsf{R}^*(n)\leqslant\begin{cases}\mathsf{M}^*(n+1) & \text{for the naive multiplication}\\ 2.5\,\mathsf{M}^*(n+1) & \text{for Karatsuba's multiplication}\end{cases}$$

and these bounds are asymptotically optimal.

Moreover when $\mathsf{M}^*(n)=K\,n\log_2(n)\log_2(\log_2(n))$ *with* $K\in\mathbb{R}_{>0}$, *one has*

$$\mathsf{R}^*(n)\sim_{n\to\infty}\mathsf{M}^*(n)\log_2(n).$$

Proof. Let $\ell:=\lfloor\log_2(n+1)\rfloor$ and $n+1=\sum_{i=0}^{\ell}\bar{n_i}\,2^i$ be the base-2 expansion of $n+1$. We can express $\mathsf{R}^*(n)$ in terms of auxiliary complexity functions by

$$\mathsf{R}^*(n)=2\,\mathsf{M}^{(2)}(n+1)-3\,\mathsf{M}^{(1)}(n+1)+\mathsf{M}^*(2^\ell). \tag{1.3}$$

Assume that M^* is the cost function of the naive or Karatsuba's multiplication. Then, using Lemma 1.16, one has

$$\begin{aligned}\mathsf{R}^*(n) &= 2\left(\frac{2^\alpha}{2^\alpha-2}\sum_{i=0}^{\ell}\bar{n_i}\,\mathsf{M}^*(2^i)-\frac{2\,n}{2^\alpha-2}\right)-3\left(\frac{2^\alpha}{2^\alpha-1}\,\mathsf{M}^*(2^\ell)-\frac{1}{2^\alpha-1}\right)+\mathsf{M}^*(2^\ell)\\ &= \left(\frac{2\cdot2^\alpha}{2^\alpha-2}-\frac{3\cdot2^\alpha}{2^\alpha-1}+1\right)\mathsf{M}^*(2^\ell)+\frac{2\cdot2^\alpha}{2^\alpha-2}\sum_{i=0}^{\ell-1}\bar{n_i}\,\mathsf{M}^*(2^i)+\frac{3}{2^\alpha-1}-\frac{4\,n}{2^\alpha-2}\\ &= C_1\,\mathsf{M}^*(2^\ell)+C_2\sum_{i=0}^{\ell-1}\bar{n_i}\,\mathsf{M}^*(2^i)-C_3\end{aligned}$$

with $C_1=\frac{2^\alpha+2}{(2^\alpha-2)(2^\alpha-1)}$, $C_2=\frac{2\cdot2^\alpha}{2^\alpha-2}$ and $C_3=\frac{4\,n}{2^\alpha-2}-\frac{3}{2^\alpha-1}$. We begin by proving that for all $n\in\mathbb{N}_{>0}$, $C_3\geqslant0$. Indeed $(C_3\geqslant0)\Leftrightarrow\left(n\geqslant\frac{3(2^\alpha-2)}{4(2^\alpha-1)}\right)$ and

$$\frac{3(2^\alpha-2)}{2(2^\alpha-1)}=\begin{cases}1/2 & \text{for } \alpha=2 \quad \text{(naïve multiplication)}\\ 3/8 & \text{for } \alpha=\log_2(3) \quad \text{(Karatsuba's multiplication)}\end{cases}.$$

We can use Lemma 1.17 to deduce that $\mathsf{R}^*(n)\leqslant C_1\,\mathsf{M}^*(n+1)$ because $C_2/C_1\leqslant(\mathsf{M}^*(3)-\mathsf{M}^*(2))$ for both naïve and Karatsuba's multiplication.

For $n=2^m$, one has

$$\mathsf{R}^*(2^m) = C_1\,\mathsf{M}^*(2^m)-C_3\sim_{m\to\infty}C_1\,\mathsf{M}^*(2^m).$$

The result for FFT multiplication is a consequence of Lemma 1.18 and Equation (1.3). □

The previous proposition proves the first row in the second table given in the introduction of this section.

1.3.5 Relaxed multiplication with middle and short products

In this subsection, we introduce a *new on-line algorithm* that uses both middle and short products. This algorithm improves by a constant factor the relaxed multiplication of the previous subsection.

We start by giving an overview of our scheme of computation. Figure 1.7 sums up the computations of the relaxed product algorithm using middle and short products.

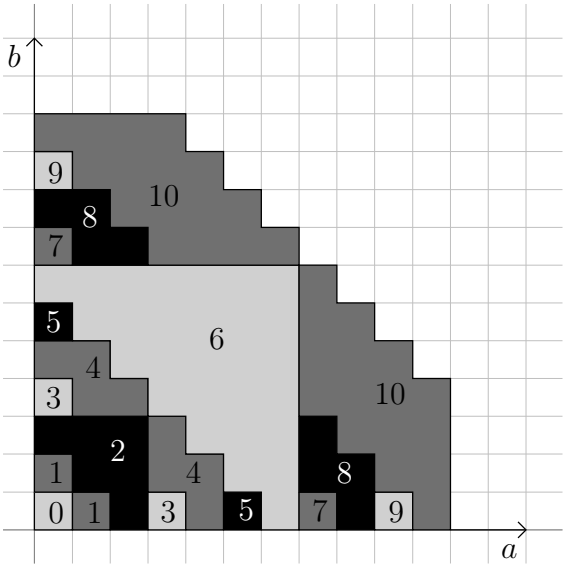

Figure 1.7. Relaxed multiplication with middle and short products

Similarly to the classical relaxed multiplication, we build our new relaxed multiplication algorithm on top of the semi-relaxed product with middle algorithm. The construction is recursive. Suppose that you have a relaxed multiplication algorithm up to precision $2^m - 1$ and that all the coefficients

$$\sum_{i=0}^{2^m-2} \sum_{j=0}^{2^m-2} a_i\, b_j\, p^{i+j}$$

of the product were computed at step $2^m - 2$. Then, for steps i with $2^m - 1 \leqslant i \leqslant 2^{m+1} - 3$, we perform two semi-relaxed products for computing $a_{2^m-1\ldots\infty}\ b$ and $a\ b_{2^m-1\ldots\infty}$. Therefore, at step $2^{m+1} - 3$, we have computed the coefficients $\sum_{0\leqslant i+j\leqslant 2^{m+1}-3} a_i\ b_j\ p^{i+j}$ of $a\ b$. In order to continue the induction, we have to compute at step $d = 2^{m+1} - 2$ the missing terms

$$\sum_{0\leqslant i,j\leqslant d,\ i+j\geqslant d} a_i\, b_j\, p^{i+j}.$$

These terms form a triangle on the diagram and can be computed by a short product

$$\sum_{0\leqslant i,j\leqslant d,\ i+j\geqslant d} a_i\, b_j\, p^{i+j} := \mathrm{rev}_d(\mathrm{SP}(\mathrm{rev}_d(a), \mathrm{rev}_d(b)))\ p^d$$

where $\mathrm{rev}_d(a) = \sum_{i=0}^{d} a_{d-i}\, p^i$. Thus, right after step $2^{m+1}-2$, we have the terms $\sum_{i=0}^{2^{m+1}-2} \sum_{j=0}^{2^{m+1}-2} a_i\, b_j\, p^{i+j}$ of the product $a\, b$ and we can pursue the induction. This gives us the following algorithm, that is on-line with respect to both inputs a and b.

Algorithm `RelaxedProductMiddleStep`

Input: $a, b, c \in R_p$ and $i \in \mathbb{N}$
Output: $c \in R_p$

1. $m = \nu_2(i+2)$
2. **if** $(i+2 = 2^m)$
   a. $c = c + \mathrm{rev}_i(\mathrm{SP}(\mathrm{rev}_i(a_{0\dots i+1}), \mathrm{rev}_i(b_{0\dots i+1})))\, p^i$
   b. **return** c
3. $c = c + \mathrm{MP}(a_{i-2^m+1\dots i+1}, b_{0\dots 2^{m+1}-1})$
4. $c = c + \mathrm{MP}(b_{i-2^m+1\dots i+1}, a_{0\dots 2^{m+1}-1})$
5. **return** c

Remark 1.25. Even if there is no efficient short FFT multiplication algorithm, we can compute the short product of Step 2 efficiently. Indeed, we noticed in Section 1.2.3 that we adapt the FFT multiplication to compute $c_{0\dots n} + c_{n\dots 2n-1}$ where $c = a\, b$ and a, b are polynomials of length n. Since the part $c_{0\dots n}$ was already computed by previous steps, we can access to $c_{n\dots 2n-1} = \mathrm{rev}_{n-1}(\mathrm{SP}(\mathrm{rev}_{n-1}(a_{0\dots n}), \mathrm{rev}_{n-1}(b_{0\dots n})))$ in half the time of a multiplication.

As expected, our algorithm is a relaxed algorithm that computes the product of two elements $a, b \in R_p$. These properties can be read on Figure 1.7.

Proposition 1.26. *Algorithm* `RelaxedProductMiddleStep` *satisfies Property* $(\mathcal{OL})$.

Complexity analysis Denote by $\mathsf{RM}^*(n)$ the cost of the relaxed multiplication with middle products up to precision n. Let $\ell := \lfloor \log_2(n+1) \rfloor$ so that this costs is given by

$$\mathsf{RM}^*(n) = \sum_{k=1}^{\ell} \mathsf{SP}^*(2^k - 1) + 2 \sum_{k=0}^{\ell-1} \left\lfloor \frac{n+1}{2^{k+1}} - \frac{1}{2} \right\rfloor \mathsf{MP}^*(2^k).$$

This formula comes from the fact that two middle products in size 2^k are done every 2^{k+1} steps, starting from step $3 \cdot 2^k - 2$. We distinguish two cases for Karatsuba's multiplication depending on the value of the ratio C_{SP} between short and plain multiplication.

Proposition 1.27. *One has*

$$\mathsf{RM}^*(n) \leqslant \begin{cases} \mathsf{M}^*(n+1) & \text{for the naive multiplication with } C_{\mathsf{SP}} = 1/2 \\ 1.75\, \mathsf{M}^*(n+1) & \text{for Karatsuba's multiplication if } C_{\mathsf{SP}} = 1/2 \\ 2.5\, \mathsf{M}^*(n+1) & \text{for Karatsuba's multiplication if } C_{\mathsf{SP}} = 1 \end{cases}$$

and these bounds are asymptotically optimal.

Moreover, when $\mathsf{M}^*(n) = K\, n \log_2(n) \log_2(\log_2(n))$ *with* $K \in \mathbb{R}_{>0}$, *one has*

$$\mathsf{RM}^*(n) \sim_{n\to\infty} \frac{1}{2}\,\mathsf{M}^*(n)\log_2(n).$$

As we will see in the following proof, the supremum of the ratio $\mathsf{RM}^*(n)/\mathsf{M}^*(n+1)$ depends linearly in C_{SP}. Therefore we can deduce this supremum for other C_{SP}. For example, in our implementation, we use the hybrid Karatsuba/naïve algorithm for plain multiplication (see Section 1.2.3) and an odd/even decomposition for short product. In this situation, the short product has a ratio $C_{\mathsf{SP}} = 0.6$. Although Proposition 1.27 does not deal with this hybrid multiplication algorithm, we believe the results for "pure" Karatsuba's multiplication should apply in this case for n large enough and yield a bound $\mathsf{RM}^*(n) \leqslant 1.9\,\mathsf{M}^*(n)$.

Proof. Let $\ell := \lfloor \log_2(n+1) \rfloor$ and $n+1 = \sum_{i=0}^{\ell} \bar{n_i}\, 2^i$ be the base-2 expansion of $n+1$. Since $\mathsf{MP}^*(n) = \mathsf{M}^*(n)$, we can express $\mathsf{RM}^*(n)$ in terms of auxiliary complexity functions by

$$\begin{aligned}\mathsf{RM}^*(n) &\leqslant C_{\mathsf{SP}} \sum_{k=1}^{\ell} \mathsf{M}^*(2^k-1) + 2\sum_{k=0}^{\ell-1} \left(\left\lfloor \frac{n+1}{2^{k+1}} + \frac{1}{2} \right\rfloor - 1 \right) \mathsf{M}^*(2^k) \\ &\leqslant (C_{\mathsf{SP}} - 2)\,\mathsf{M}^{(1)}(n+1) + 2\,\mathsf{M}^{(3)}(n+1)\end{aligned}$$

Assume that M^* is the cost function of the naive or Karatsuba's multiplication. Then, using Lemma 1.16, one has

$$\begin{aligned}\mathsf{R}^*(n) &= (C_{\mathsf{SP}} - 2)\left(\frac{2^\alpha}{2^\alpha - 1}\,\mathsf{M}^*(2^\ell) - \frac{1}{2^\alpha - 1} \right) + 2\left(\frac{2^\alpha - 1}{2^\alpha - 2} \sum_{i=0}^{\ell} \bar{n_i}\,\mathsf{M}^*(2^i) - \frac{n}{2^\alpha - 2} \right) \\ &= C_1\,\mathsf{M}^*(2^\ell) + C_2 \sum_{i=0}^{\ell-1} \bar{n_i}\,\mathsf{M}^*(2^i) - C_3\end{aligned}$$

with $C_1 = \frac{2^\alpha(2^\alpha-2)\,C_{\mathsf{SP}} + 2}{(2^\alpha-2)(2^\alpha-1)}$, $C_2 = \frac{2\cdot(2^\alpha-1)}{2^\alpha-2}$ and $C_3 = \frac{2\,n}{2^\alpha-2} - \frac{2-C_{\mathsf{SP}}}{2^\alpha-1}$. We begin by noticing that for all $n \in \mathbb{N}_{>0}$, $C_3 \geqslant 0$. Indeed $(C_3 \geqslant 0) \Leftrightarrow \left(n \geqslant \frac{(2-C_{\mathsf{SP}})(2^\alpha-2)}{2(2^\alpha-1)} \right)$ and since $C_{\mathsf{SP}} \geqslant 1/2$, one has

$$\frac{(2-C_{\mathsf{SP}})(2^\alpha-2)}{2(2^\alpha-1)} \leqslant \begin{cases} 1/2 & \text{for } \alpha = 2 \quad \text{(naïve multiplication)} \\ 3/8 & \text{for } \alpha = \log_2(3) \quad \text{(Karatsuba's multiplication)} \end{cases}.$$

We can use Lemma 1.17 to deduce that $\mathsf{R}^*(n) \leqslant C_1\,\mathsf{M}^*(n+1)$ because $C_2/C_1 \leqslant (\mathsf{M}^*(3) - \mathsf{M}^*(2))$ for both naïve and Karatsuba's multiplication and any constant $1/2 \leqslant C_{\mathsf{SP}} \leqslant 1$.

These bounds are asymptotically optimal:

$$\mathsf{R}^*(2^m) = C_1\,\mathsf{M}^*(2^m) - C_3 \sim_{m\to\infty} C_1\,\mathsf{M}^*(2^m).$$

Lemma 1.18 also gives the result for FFT multiplication. □

1.3.6 Block variant

For large n, the ratio between on-line and off-line multiplication algorithms can get too big to be of any interest. This happens usually when using the FFT multiplication, as the ratio grows like $\log_2(n)$.

In this case, a d-block variant of an algorithm uses a p^d-adic representation of the p-adics in $R_{(p^d)} = R_{(p)}$. Instead of writing $y = \sum_{n\geqslant 0} y_n\, p^n$, we write $y = \sum_{n\geqslant 0} (y_n + y_{n+1}\, p + \cdots + y_{n+d-1}\, p^{d-1})\, p^{dn} \in R_{(p^d)}$. Then the d-block variant algorithm is on-line in the p^d-adic representation. It means that it computes d new coefficients at each step, instead of one coefficient at a time for an on-line algorithm in the p-adic representation.

By doing so, we can decrease the ratio between on-line and off-line multiplication algorithms by a constant; a complexity for relaxed product that was like $\mathsf{M}^*(n)\log_2(n)$ in p-adic representation gives a new complexity $\mathsf{M}^*(n)\log_2(n/d)$ in p^d-adic representation. We refer to [BHL11] for details.

1.4 Implementation and timings

We give timings, in seconds, of the different multiplication algorithms for the case of power series $\mathbb{F}_p[[X]]$ with the 29-bit prime number $p = 268435459$. Computations were done on one core of a INTEL CORE i5 at 2.40 GHz with 4Gb of RAM running a 32-bit LINUX. Our implementation uses the polynomial multiplication of NTL 5.5.2 [S+90]. The threshold between the naive and Karatsuba's multiplications is at degree 16 and the one between Karatsuba's and FFT multiplications at degree 1500.

In Figure 1.8, we plot the timings of the multiplication of polynomials and of several relaxed multiplication algorithms on power series depending on the precision in abscissa. Both coordinate axes use a logarithmic scale. The name `SRM` stands for the semi-relaxed multiplication using middle product of Section 1.3.3. The name `RM` stands for the relaxed multiplication using middle (and short) product of Section 1.3.5. And so on.

We can see that polynomial multiplication is faster from precision 8 on. The gap between any relaxed algorithm and the polynomial product remains constant in the Karatsuba range and grows as soon as we reach the FFT multiplication.

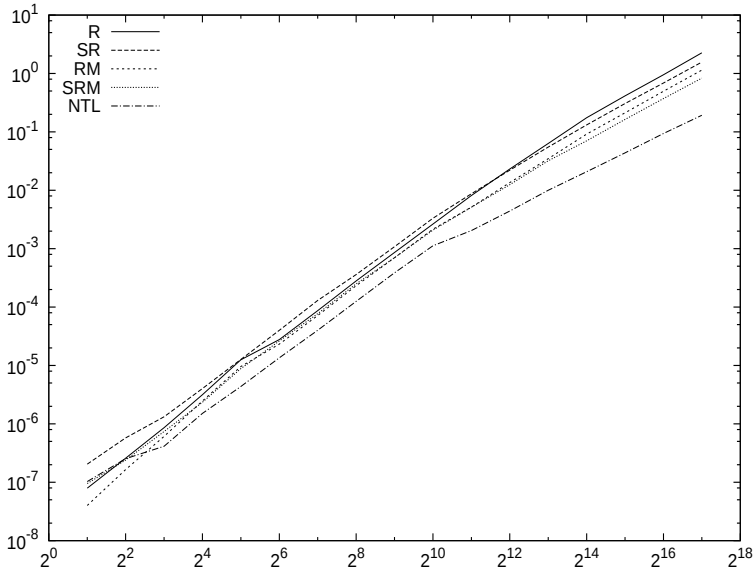

Figure 1.8. Timings of different multiplication algorithms

In Figure 1.9, we display the ratio of timings of several relaxed multiplication algorithms compared to the polynomial product depending on the precision in abscissa. This plot confirms the theoretical bounds for Karatsuba's multiplication, except on a few points, and the constants 1, 1/2 or 1/4 in the asymptotic equivalents for the FFT multiplication. We can see that the use of middle product always improves the performance of both the relaxed and semi-relaxed multiplication algorithms. We save up to a factor 2, which is attained for the FFT multiplication.

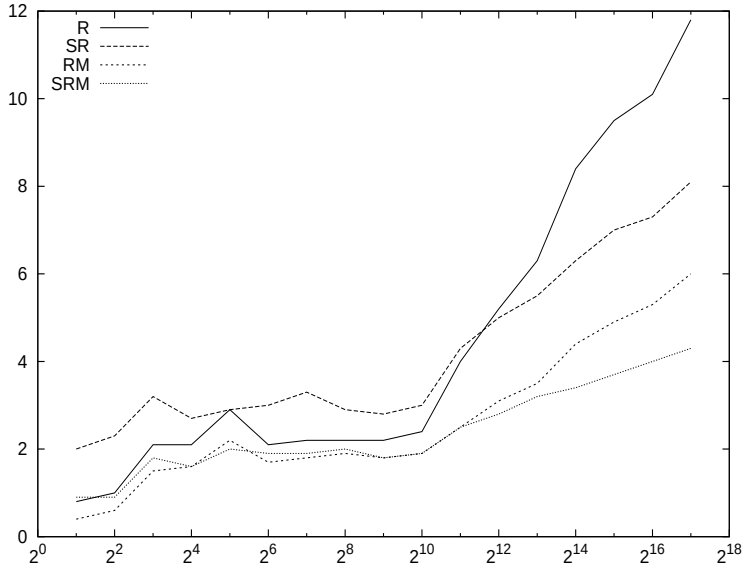

Figure 1.9. Ratio of timings of different relaxed products w.r.t. polynomial multiplication

Chapitre 2
Recursive p-adics

This chapter is based on a section of the paper *Relaxed Hensel p-adic lifting of algebraic systems* published with J. Berthomieu in the proceedings of *ISSAC'12* [BL12]. The present chapter contains additional details, proofs and examples.

One strength of relaxed algorithms is to allow the computation of recursive p-adics. The contribution of this chapter is to give a precise framework, based on our notion of shifted algorithms, to compute recursive p-adics. The *main result*, Proposition 2.17, is the building block of almost all relaxed algorithms in this thesis. Most of the following chapters are dedicated to the exploration of the consequences of this framework to further problems.

As we will see, solving a recursive equation is very similar to verifying it. Therefore, the cost of solving such an equation depends mainly on the cost of evaluating the equation.

2.1 Straight-line programs

Straight-line programs are a model of computation that consist in ordered lists of instructions without branching. We give a short presentation of this notion and refer to [BCS97] for more details. We will use this model of computation to describe and analyze the forthcoming recursive operators and shifted algorithms.

Let R be a ring and A an R-algebra. A *straight-line program* (s.l.p.) is an ordered sequence of operations between elements of A. An *operation* of *arity* r is a map from a subset $\mathcal{D}$ of A^r to A. We usually work with the binary arithmetic operators $+, -, \cdot\colon \mathcal{D} = A^2 \to A$. We also define for $r \in R$ the 0-ary operations r^c whose output is the constant r and the unary scalar multiplication $r \times \_$ by r. We denote the set of all these operations by R^c and R. Let us fix a set of operations Ω, usually $\Omega = \{+, -, \cdot\} \cup R \cup R^c$.

An s.l.p. starts with a number ℓ of *input* parameters indexed from $-(\ell-1)$ to 0. It has L *instructions* $\Gamma_1, \ldots, \Gamma_L$ with $\Gamma_i = (\omega_i; u_{i,1}, \ldots, u_{i,r_i})$ where $-\ell < u_{i,1}, \ldots, u_{i,r_i} < i$ and r_i is the arity of the operation $\omega_i \in \Omega$. The s.l.p. Γ is *executable* on $a = (a_0, \ldots, a_{\ell-1})$ with *result sequence* $b = (b_{-\ell+1}, \ldots, b_L) \in A^{\ell+L}$, if $b_i = a_{\ell-1+i}$ whenever $-(\ell-1) \leqslant i \leqslant 0$ and $b_i = \omega_i(b_{u,1}, \ldots, b_{u,r_i})$ with $(b_{u,1}, \ldots, b_{u,r_i}) \in \mathcal{D}_{\omega_i}$ whenever $1 \leqslant i \leqslant L$. We say that the s.l.p. Γ *computes* $b \in A$ on the entries $a_1, \ldots, a_\ell$ if Γ is executable on $a_1, \ldots, a_\ell$ over A and b is a member of the result sequence.

The *multiplicative complexity* $L^*(\Gamma)$ of an s.l.p. Γ is the number of operations ω_i that are multiplications $\cdot$ between elements of A.

Example 2.1. Let $R = \mathbb{Z}$, $A = \mathbb{Z}[X, Y]$ and Γ be the s.l.p. with two input parameters indexed -1, 0 and

$$\Gamma_1 = (\cdot; -1, -1), \quad \Gamma_2 = (\cdot; 1, 0), \quad \Gamma_3 = (1^c), \quad \Gamma_4 = (-; 2, 3), \quad \Gamma_5 = (3 \times \_; 1).$$

First, its multiplicative complexity is $L^*(\Gamma) = 2$. Then, Γ is executable on $(X, Y) \in A^2$, and for this input its result sequence is $(X, Y, X^2, X^2\,Y, 1, X^2\,Y - 1, 3\,X^2)$.

Remark 2.2. For the sake of simplicity, we will associate a "canonical" arithmetic expression with an s.l.p. It is the same operation as when one writes an arithmetic expression in a programming language, e.g. C, and a compiler turns it into an s.l.p. In our case, we fix an arbitrary compiler that starts by the left-hand side of an arithmetic expression. We use the binary powering algorithm to compute powers of an expression.

For example, the arithmetic expression $\varphi\colon Z \mapsto Z^4 + 1$ can be represented by the s.l.p. with one argument and instructions

$$\Gamma_1 = (\cdot; 0, 0), \quad \Gamma_2 = (\cdot; 1, 1), \quad \Gamma_3 = (1^c), \quad \Gamma_4 = (+; 2, 3).$$

2.2 Recursive p-adics

The study of on-line algorithms is motivated by its efficient implementation of recursive p-adics. To the best of our knowledge, the paper [Wat89] was the first to mention the lazy computation of power series which are solutions of a fixed point equation $y = \Phi(y)$. The paper [Hoe02], in addition to rediscovering the fast on-line multiplication algorithm of [FS74], connected for the first time this fast multiplication algorithm to the on-line computation of recursive power series. Van der Hoeven named these on-line algorithms, that use the fast on-line multiplication, *relaxed algorithms*. Article [BHL11] generalizes relaxed algorithms for p-adics.

We contribute by clarifying the setting in which recursive p-adics can be computed from their fixed point equations $y = \Phi(y)$ by an on-line algorithm. For this matter, we introduce the notion of *shifted algorithm*.

We will work with recursive p-adics in a simple case and do not need the general context of recursive p-adics [Kap01, Definition 7]. We denote by $\nu_p(a)$ the valuation in p of the p-adic a. For vectors or matrices $A \in \mathcal{M}_{r\times s}(R_p)$, we define $\nu_p(A) := \min_{i,j}(\nu_p(A_{i,j}))$. We start by giving a definition of recursive p-adics and their recursive equation that suits our needs.

Definition 2.3. *Let $\ell \in \mathbb{N}$, $\boldsymbol{\Phi} \in (R_p[Y_1, \ldots, Y_\ell])^\ell$, $\boldsymbol{y} \in (R_p)^\ell$ be a fixed point of $\boldsymbol{\Phi}$,* i.e. *$\boldsymbol{y} = \boldsymbol{\Phi}(\boldsymbol{y})$. We write $\boldsymbol{y} = \sum_{i\in\mathbb{N}} \boldsymbol{y}_i\, p^i$ the p-adic decomposition of $\boldsymbol{y}$. Let us denote $\boldsymbol{\Phi}^0 = \mathrm{Id}$ and, for all $n \in \mathbb{N}^*$, $\boldsymbol{\Phi}^n = \boldsymbol{\Phi} \circ \cdots \circ \boldsymbol{\Phi}$ (n times).*

Then, we say that the coordinates $(y_1, \ldots, y_\ell)$ of $\boldsymbol{y}$ are recursive p-adics *and that the recursive operator $\boldsymbol{\Phi}$* allows the computation *of $\boldsymbol{y}$ if, for all $n \in \mathbb{N}$, we have $\nu_p(\boldsymbol{y} - \boldsymbol{\Phi}^n(\boldsymbol{y}_0)) \geqslant n + 1$.*

The general case with more initial conditions $\boldsymbol{y}_0, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_s$ is not considered here but we believe it would to be an interesting extension of these results.

Proposition 2.4. *Let $\boldsymbol{\Phi} \in (R_p[Y_1, \ldots, Y_\ell])^\ell$ with a fixed point $\boldsymbol{y} \in R_p^\ell$ and let $\boldsymbol{y}_0 = \boldsymbol{y} \bmod p$. Suppose $\nu_p(\mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{y}_0)) > 0$. Then $\boldsymbol{\Phi}$ allows the computation of $\boldsymbol{y}$.*

Moreover, for all $n \leqslant m \in \mathbb{N}^*$, the p-adic coefficient $(\boldsymbol{\Phi}(\boldsymbol{y}))_n$ does not depend on the coefficient $\boldsymbol{y}_m$, i.e. *$(\boldsymbol{\Phi}(\boldsymbol{y}))_n = (\boldsymbol{\Phi}(\boldsymbol{y} + \boldsymbol{a}))_n$ for any $\boldsymbol{a} \in (p^n\, R_p)^\ell$.*

Proof. We prove by induction on n that $\nu_p(\boldsymbol{y}-\boldsymbol{\Phi}^n(\boldsymbol{y}_0)) \geqslant n+1$. First, notice that $\nu_p(\boldsymbol{y}-\boldsymbol{y}_0) \geqslant 1$. Let us prove the claim for $n+1$, assuming that it is verified for n. For all $\boldsymbol{y}, \boldsymbol{z} \in R_p^\ell$, there exists, by Taylor expansion of $\boldsymbol{\Phi}$ at $\boldsymbol{z}$, vectors of p-adics $\boldsymbol{\Theta}_{i,j}(\boldsymbol{y},\boldsymbol{z}) \in R_p^\ell$ for $1 \leqslant i \leqslant j \leqslant \ell$ such that

$$\boldsymbol{\Phi}(\boldsymbol{y})-\boldsymbol{\Phi}(\boldsymbol{z}) = \mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{z})\,(\boldsymbol{y}-\boldsymbol{z}) + \sum_{1\leqslant i\leqslant j\leqslant \ell} (y_i-z_i)\,(y_j-z_j)\,\boldsymbol{\Theta}_{i,j}(\boldsymbol{y},\boldsymbol{z}).$$

For all $n \in \mathbb{N}$, we set $\boldsymbol{y}_{(n)} := \boldsymbol{\Phi}^n(\boldsymbol{y}_0)$ and we apply the previous statement to $\boldsymbol{y}$ itself and $\boldsymbol{z} = \boldsymbol{y}_{(n)}$:

$$\begin{aligned} \boldsymbol{y}-\boldsymbol{y}_{(n+1)} &= \boldsymbol{\Phi}(\boldsymbol{y})-\boldsymbol{\Phi}(\boldsymbol{y}_{(n)}) \\ &= \mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{y}_{(n)})\,(\boldsymbol{y}-\boldsymbol{y}_{(n)}) + \sum_{1\leqslant i\leqslant j\leqslant \ell} (y_i-y_{(n),i})\,(y_j-y_{(n),j})\,\boldsymbol{\Theta}_{i,j}(\boldsymbol{y},\boldsymbol{y}_{(n)}). \end{aligned}$$

By the induction hypothesis, $\nu_p(\boldsymbol{y}-\boldsymbol{y}_{(n)}) \geqslant n+1$. Also $\nu_p(\mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{y}_{(0)})) > 0$ implies $\nu_p(\mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{y}_{(n)})) > 0$. As a consequence, one has $\nu_p(\boldsymbol{y}-\boldsymbol{y}_{(n+1)}) \geqslant n+2$.

For the second point, remark that if $\boldsymbol{a} \in (p^n R_p)^\ell$, then

$$\boldsymbol{\Phi}(\boldsymbol{y}+\boldsymbol{a})-\boldsymbol{\Phi}(\boldsymbol{y}) = \mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{y})\,\boldsymbol{a} + \sum_{1\leqslant i\leqslant j\leqslant \ell} a_i\,a_j\,\boldsymbol{\Theta}_{i,j}\,(\boldsymbol{y}+\boldsymbol{a},\boldsymbol{y}) \in (p^{n+1}R_p)^\ell$$

since $\nu_p(\mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{y})) > 0$ and $\nu_p(a_i) > 0$ because $n \in \mathbb{N}^*$. □

On-line computation of recursive p-adics Let us recall the ideas to compute $\boldsymbol{y}$ from $\boldsymbol{\Phi}(\boldsymbol{y})$ in the on-line framework. Let $\boldsymbol{\Phi}$ be given as an s.l.p. with operations in $\Omega = \{+,-,\cdot\} \cup R \cup R^c$. First, if $\boldsymbol{a} \in R_p^\ell$, we evaluate $\boldsymbol{\Phi}(\boldsymbol{a})$ in an on-line manner by performing the arithmetic operations of the s.l.p. $\boldsymbol{\Phi}$ with on-line algorithms. Let OnlineAddStep (resp. OnlineMulStep) be the step of any on-line addition (resp. multiplication) algorithm.

Algorithm `OnlineEvaluationStep`

Input: an s.l.p. $\boldsymbol{\Phi}$, $\boldsymbol{a} = (a_1,\ldots,a_\ell) \in (R_p)^\ell$, $\left[c_1^{(0)},\ldots,c_L^{(0)}\right] \in (R_p)^L$ and $i \in \mathbb{N}$
Output: $[c_1,\ldots,c_L] \in (R_p)^L$

1. $[c_{-\ell+1},\ldots,c_0] = [a_1,\ldots,a_\ell]$
2. $[c_1,\ldots,c_L] = \left[c_1^{(0)},\ldots,c_L^{(0)}\right]$
3. **for** j from 1 to L
   if $(\Gamma_j = (\text{'}+\text{'};u,v))$
   $c_j = \texttt{OnlineAddStep}(c_u, c_v, c_j, i)$
   if $(\Gamma_j = (\text{'}-\text{'};u,v))$
   $c_j = \texttt{OnlineAddStep}(c_u, -c_v, c_j, i)$
   if $(\Gamma_j = (\text{'}\cdot\text{'};u,v))$
   $c_j = \texttt{OnlineMulStep}(c_u, c_v, c_j, i)$
   if $(\Gamma_j = (r \times \_;u))$
   $c_j = \texttt{OnlineMulStep}(r, c_u, c_j, i)$
   if $(\Gamma_j = (r;))$
   $c_j = r$
4. **return** $[c_1,\ldots,c_L]$

We see that Algorithm `OnlineEvaluationStep` computes the result sequence $[c_1, ..., c_L] \in (R_p)^L$ of the s.l.p. $\mathbf{\Phi}$ on the input $\boldsymbol{a} \in (R_p)^\ell$. If one wants to evaluate $\mathbf{\Phi}$ on $\boldsymbol{a}$, it remains to loop on Algorithm `OnlineEvaluationStep`.

Algorithm `OnlineEvaluation`

Input: an s.l.p. $\mathbf{\Phi}$, $\boldsymbol{a} = (a_1, ..., a_\ell) \in (R_p)^\ell$ and $N \in \mathbb{N}$
Output: $[c_1, ..., c_L] \in (R_p)^L$

1. $[c_1, ..., c_L] = [0, ..., 0]$
2. **for** i from 0 to N
   $[c_1, ..., c_L] = \texttt{OnlineEvaluationStep}(\mathbf{\Phi}, \boldsymbol{a}, [c_1, ..., c_L], i)$
3. **return** $[c_1, ..., c_L]$

As expected, `OnlineEvaluation` is an on-line algorithm.

Proposition 2.5. *For any $N \in \mathbb{N}$, any s.l.p. $\mathbf{\Phi}$ and $\boldsymbol{a} \in (R_p)^\ell$, the output $[c_1, ..., c_L]$ of* `OnlineEvaluation`$(\mathbf{\Phi}, \boldsymbol{a}, N)$ *coincides at precision $N+1$ with the result sequence of the s.l.p. $\mathbf{\Phi}$ on the input $\boldsymbol{a}$.*

Moreover, Algorithm `OnlineEvaluation`$(\mathbf{\Phi}, \boldsymbol{a}, N)$ *is on-line with respect to its input $\boldsymbol{a}$.*

Now that we have this algorithm, we want to use the relation $\boldsymbol{y} = \mathbf{\Phi}(\boldsymbol{y})$ to compute the recursive p-adics $\boldsymbol{y}$. What we really compute is $\mathbf{\Phi}(\boldsymbol{y})$: suppose that we are at the point where we know the p-adic coefficients $\boldsymbol{y}_0, ..., \boldsymbol{y}_{N-1}$ of $\boldsymbol{y}$ and $\mathbf{\Phi}(\boldsymbol{y})$ has been computed up to its $(N-1)$st coefficient. Since in the on-line framework, the computation is done step by step, one can naturally ask for one more step of the computation of $\mathbf{\Phi}(\boldsymbol{y})$. Also, from Proposition 2.4, $(\mathbf{\Phi}(\boldsymbol{y}))_N$ depends only on $\boldsymbol{y}_0, ..., \boldsymbol{y}_{N-1}$ so that we should be able to compute it and deduce $\boldsymbol{y}_N = (\mathbf{\Phi}(\boldsymbol{y}))_N$.

We denote by $i_1, ..., i_\ell$ the indices of the outputs of the s.l.p. $\mathbf{\Phi}$.

Algorithm `OnlineRecursivePadic`

Input: an s.l.p. $\mathbf{\Phi}$, $\boldsymbol{y}_0 \in M^\ell$ and $N \in \mathbb{N}$
Output: $\boldsymbol{a} \in (R_p)^\ell$

1. $\boldsymbol{a} = \boldsymbol{y}_0$
2. $[c_1, ..., c_L] = [0, ..., 0]$
3. **for** i from 0 to N
   a. $[c_1, ..., c_L] = \texttt{OnlineEvaluationStep}(\mathbf{\Phi}, \boldsymbol{a}, [c_1, ..., c_L], i)$
   b. $\boldsymbol{a} = [c_{i_1}, ..., c_{i_\ell}]$
4. **return** $\boldsymbol{a}$

One's hope is that, with the notations of Definition 2.3, the output $\boldsymbol{a}$ of Algorithm `OnlineRecursivePadic` coincides with the recursive p-adic $\boldsymbol{y}$ at precision $N+1$. But one has to be cautious because, even if $(\mathbf{\Phi}(\boldsymbol{y}))_N$ does not depend on $\boldsymbol{y}_N$, the coefficient $\boldsymbol{y}_N$ could still be involved in anticipated computations at step N and may introduce mistakes in the following coefficients.

Here is an example of this issue that has never been raised before.

Warning 2.6. Take $R=\mathbb{Q}[X]$ and $p=X$ so that $R_p=\mathbb{Q}[[X]]$. Let Φ associated to the arithmetic expression $Y\mapsto Y^2+X$, that is the s.l.p. with one input and output and instructions

$$\Gamma_1=(\cdot;0,0),\quad \Gamma_2=(X^c),\quad \Gamma_3=(+;1,2).$$

Let y be the only fixed point of Φ satisfying $y_0=0$, that is

$$y=\frac{\sqrt{1-4\,X}-1}{2}=X+X^2+2\,X^3+5\,X^4+\mathcal{O}(X^5).$$

Since $\Phi'(0)=0$, Φ allows the computation of y.

Let us specialize Algorithm `OnlineRecursivePadic` in our case. We choose to take Algorithm `LazyAddStep` for the addition and Algorithm `RelaxedProductStep` for multiplication.

Algorithm 2.1
Input: $N\in\mathbb{N}$ **Output:** $a\in R_p$
1. $a=0\,(=y_0)$
2. $c=0$
3. **for** i from 0 to N
a. $c=$`RelaxedProductStep`(c,a,a,i)
b. $a=$`LazyAddStep`(a,c,X,i)
4. **return** a

Since we already have $a_0=y_0$ before step 0, the purpose of this step is to initialize the computations. Then at the first step, we do the computations

$$c=c+2\,a_0\,a_1\,X=0,\quad a=a+(c_1+1)\,X=X.$$

So after step 1, we get $a_1=1$, which is correct, *i.e.* $a_1=y_1$. Now at step 2 we know that a_0 and a_1 are correct and we do

$$c=c+(2\,a_0\,a_2+(a_1+a_2\,X)^2)\,X^2,\quad a=a+c_2\,X^2.$$

Even if $a_2\neq y_2$, the computations produce $c=X^2$ and $a=X+X^2$ which is correct at precision 3. As predicted by Proposition 2.4, the incorrect coefficient $(a_2=0)\neq(y_2=1)$ at the beginning of step 2 did not impact the correct result $a_2=(\Phi(a))_2=1$ at the end. However the incorrect $a_2\neq y_2$ is involved in some anticipated computations of future terms.

An error appears at step 3: we do

$$c=c+(2\,a_0\,a_3)\,X^3,\quad a=a+c_3\,X^3.$$

This gives $c = X^2$ and $a = X + X^2$ which differs from the correct result $X + X^2 + 2\,X^3$ at precision 4.

Remark 2.7. We have just seen that `OnlineRecursivePadic` do *not* work for any on-line addition and multiplication algorithms. As it turns out, it does for lazy addition and multiplication algorithms, no matter the recursive operator $\mathbf{\Phi}$ as in Proposition 2.4. Indeed, lazy algorithms do at step N the computations for $(\mathbf{\Phi}(\boldsymbol{y}))_N$, and only them. So when we begin to compute $(\mathbf{\Phi}(\boldsymbol{y}))_N$, that is at step N, we know $\boldsymbol{y}_0, ..., \boldsymbol{y}_{N-1}$ and the unknown value of $\boldsymbol{y}_N$ does not change the result. Therefore, $(\mathbf{\Phi}(\boldsymbol{y}))_N$ is computed correctly for all $N \in \mathbb{N}$.

This may explain why the issue was not spotted before by papers dealing only with lazy algorithms [Wat89].

As a conclusion, even if $(\mathbf{\Phi}(\boldsymbol{y}))_N$ does not depend on the p-adic coefficient $\boldsymbol{y}_N$, the coefficient $\boldsymbol{y}_N$ can be involved in anticipated computations leading to errors later. Since we do not know $\boldsymbol{y}_N$ at step N, we must proceed otherwise. Given a recursive operator $\mathbf{\Phi} \in R_p[Y_1, ..., Y_\ell]^\ell$, we create another s.l.p. $\mathbf{\Psi}$ that computes the same polynomials $\mathbf{\Phi}(Y_1, ..., Y_\ell)$ but does not read the p-adic coefficient $\boldsymbol{y}_N$ at step N.

2.3 Shifted algorithms

Because of the issue raised in Warning 2.6, we need to make explicit the fact that $\boldsymbol{y}_N$ is not read at step N of the computation of $\mathbf{\Phi}(\boldsymbol{y})$. This issue was never mentioned in the literature before. In this section, we define the notion of shifted algorithms and prove that these algorithms compute correctly recursive p-adics by the on-line method of previous section.

We introduce for all s in $\mathbb{N}^*$ two new operators:

$$\begin{array}{lccc} p^s \times \_ : & R_p & \to & R_p \\ & a & \mapsto & p^s\, a, \end{array} \qquad \begin{array}{lccc} \_/p^s : & p^s\, R_p & \to & R_p \\ & a & \mapsto & a/p^s. \end{array}$$

The implementation of these operators just moves (or shifts) the coefficients of the input. It does not call any multiplication algorithm.

Algorithm `OnlineShiftStep`
Input: $a, c \in R_p$, $s \in \mathbb{Z}$ and $i \in \mathbb{N}$ **Output:** $c \in R_p$
1. $c = c + a_{i-s}\, p^i$
2. **return** c

Let Ω' be the set of operations $\{+, -, \cdot, p^s \times \_, \_/p^s\} \cup R \cup R^c$. We update the definition of Algorithm `OnlineEvaluationStep` to accept s.l.p.'s with operations in Ω'.

Algorithm `OnlineEvaluationStep`

Input: an s.l.p. $\mathbf{\Phi}$, $\boldsymbol{a} = (a_1, ..., a_\ell) \in (R_p)^\ell$, $\left[c_1^{(0)}, ..., c_L^{(0)}\right] \in (R_p)^L$ and $i \in \mathbb{N}$
Output: $[c_1, ..., c_L] \in (R_p)^L$

1. $[c_{-\ell+1}, ..., c_0] = [a_1, ..., a_\ell]$
2. $[c_1, ..., c_L] = \left[c_1^{(0)}, ..., c_L^{(0)}\right]$
3. **for** j from 1 to L
   if $(\Gamma_j = (\text{'}+\text{'}; u, v))$
   $c_j = \texttt{OnlineAddStep}(c_u, c_v, c_j, i)$
   if $(\Gamma_j = (\text{'}-\text{'}; u, v))$
   $c_j = \texttt{OnlineAddStep}(c_u, -c_v, c_j, i)$
   if $(\Gamma_j = (\text{'}\cdot\text{'}; u, v))$
   $c_j = \texttt{OnlineMulStep}(c_u, c_v, c_j, i)$
   if $(\Gamma_j = (r \times \_; u))$
   $c_j = \texttt{OnlineMulStep}(r, c_u, c_j, i)$
   if $(\Gamma_j = (r;))$
   $c_j = r$
   if $(\Gamma_j = (p^s \times \_; u))$
   $c_j = \texttt{OnlineShiftStep}(c_u, c_j, s, i)$
   if $(\Gamma_j = (\_/p^s; u))$
   $c_j = \texttt{OnlineShiftStep}(c_u, c_j, -s, i)$
4. **return** $[c_1, ..., c_L]$

In the next definition, we define a number, the *shift*, that will indicate which coefficients of an input of an s.l.p. are read at any step.

Definition 2.8. *Let us consider a Turing machine* T *with* n *inputs in* Σ^* *and one output in* Δ^**, where* Σ *and* Δ *are sets. We denote by* $\boldsymbol{a} = (a^1, ..., a^\ell)$ *an input sequence of* T *and, for all* $1 \leqslant i \leqslant \ell$*, we write* $a^i = a_0^i a_1^i ... a_n^i$ *with* $a_j^i \in \Sigma$*. We denote by* $c_0 c_1 ... c_n$ *the corresponding output, where* $c_k \in \Delta$*.*

For all input index i *with* $1 \leqslant i \leqslant \ell$*, we define the* set of shifts $\mathcal{S}(\mathrm{T}, i) \subseteq \mathbb{Z}$ *as the set of integers* $s \in \mathbb{Z}$ *such that, for all input sequences* $\boldsymbol{a}$*, the Turing machine produces* c_k *before reading* a_j^i *for* $0 \leqslant k < j + s \leqslant n$*.*

Also, we define the set of shifts $\mathcal{S}(\mathrm{T}) \subseteq \mathbb{Z}$ *by*

$$\mathcal{S}(\mathrm{T}) := \bigcap_{1 \leqslant i \leqslant \ell} \mathcal{S}(\mathrm{T}, i).$$

If $s \in \mathcal{S}(\mathrm{T})$*, we say that* T *has* shift s*.*

Algorithms do not have a unique shift: if $s \in \mathcal{S}(\mathrm{T}, i)$ then $s' \in \mathcal{S}(\mathrm{T}, i)$ for all integers $s' \leqslant s$. The definition of shift for a Turing machine is a generalization of the notion of on-line algorithms.

Corollary 2.9. *A Turing machine* T *is on-line if and only if* $0 \in \mathcal{S}(\mathrm{T})$*. Its* i*th input is an on-line argument if and only if* $0 \in \mathcal{S}(\mathrm{T}, i)$*.*

Example 2.10. Let $s \in \mathbb{Z}$ and denote by $\texttt{OnlineShift}(a, c, s, N)$ the algorithm that put $\texttt{OnlineShiftStep}(a, c, s, i)$ in a loop with i varying from 0 to $N \in \mathbb{N}$. This construction is similar to Algorithm $\texttt{Loop}_{\texttt{Algo}}$ in Chapter 1. Then Algorithm $\texttt{OnlineShift}(a, c, s, N)$ has shift s with respect to its input a.

Let us now focus on the rules to compute a shift. Let $\mathbf{\Phi}$ be a s.l.p. and N be an integer. We are interested in the shift of Algorithm $\texttt{OnlineEvaluation}(\mathbf{\Phi}, \boldsymbol{a}, N)$ with respect to its p-adic input $\boldsymbol{a}$. Let $\texttt{OnlineEvaluation}(\mathbf{\Phi}, \_, N)$ denote the partial algorithm which maps $\boldsymbol{a}$ to $\texttt{OnlineEvaluation}(\mathbf{\Phi}, \boldsymbol{a}, N)$. Recall that Algorithm $\texttt{OnlineEvaluation}(\mathbf{\Phi}, \_, N)$ merely executes the operations of the s.l.p. $\mathbf{\Phi}$ with on-line algorithms. For this reason we are able to define an integer $\mathrm{sh}(\Gamma, j, h)$ for each output index j and input index h, that will be a shift of Algorithm $\texttt{OnlineEvaluation}(\mathbf{\Phi}, \_, N)$ with respect to this input and this output.

Definition 2.11. *Let $\Gamma = (\Gamma_1, \ldots, \Gamma_L)$ be an s.l.p. over the R-algebra R_p with ℓ input parameters and operations in Ω'. For any operation index j such that $-(\ell-1) \leqslant j \leqslant L$ and for any input index h such that $-(\ell-1) \leqslant h \leqslant 0$, the* shift $\mathrm{sh}(\Gamma, j, h)$ *of its jth result b_j with respect to its hth input argument is an element of $\mathbb{Z} \cup \{+\infty\}$ defined as follows.*

If j corresponds to an input, i.e. *$j \leqslant 0$, we define for all $-(\ell-1) \leqslant h \leqslant 0$*

$$\mathrm{sh}(\Gamma, j, h) = \begin{cases} 0 & \text{if } j = h \\ +\infty & \text{if } j \neq h \end{cases}.$$

If j corresponds to an operation, i.e. *$j > 0$, then for all $-(\ell-1) \leqslant h \leqslant 0$*

- *if $\Gamma_j = (\omega_j; u, v)$ with $\omega_j \in \{+, -, \cdot\}$, then we set*

$$\mathrm{sh}(\Gamma, j, h) := \min\left(\mathrm{sh}(\Gamma, u, h), \mathrm{sh}(\Gamma, v, h)\right);$$

- *if $\Gamma_j = (r^c;)$, then $\mathrm{sh}(\Gamma, j, h) := +\infty$;*
- *if $\Gamma_j = (p^s \times \_; u)$, then $\mathrm{sh}(\Gamma, j, h) := \mathrm{sh}(\Gamma, u, h) + s$;*
- *if $\Gamma_j = (\_/p^s; u)$, then $\mathrm{sh}(\Gamma, j, h) := \mathrm{sh}(\Gamma, u, h) - s$;*
- *if $\Gamma_j = (\omega; u)$ with $\omega \in R$, then we set $\mathrm{sh}(\Gamma, j, h) := \mathrm{sh}(\Gamma, u, h)$.*

Finally if Γ has r outputs indexed by $j_1, \ldots, j_r$ in the result sequence, then we define

$$\mathrm{sh}(\Gamma) := \min\left(\{\mathrm{sh}(\Gamma, j_k, h) \mid 0 \leqslant k \leqslant r, -(\ell-1) \leqslant h \leqslant 0\}\right).$$

The following proposition proves that Algorithm $\texttt{OnlineEvaluation}(\Gamma, \_, N)$ has shift $\mathrm{sh}(\Gamma, j, h)$ with respect to its hth input and its jth output.

Proposition 2.12. *With the notations of Definition 2.11, let $\boldsymbol{y} = (y_0, \ldots, y_{\ell-1}) \in (R_p)^\ell$ be such that Γ is executable on input $\boldsymbol{y}$. Let $N \in \mathbb{N}$ and $c_1, \ldots, c_L$ be the output of $\texttt{OnlineEvaluation}(\Gamma, \boldsymbol{y}, N)$. Let $0 \leqslant h < \ell$ and $\bar{h} = h - (\ell-1)$ be the index of the input y_h in the result sequence.*

Then, the computation of $(c_j)_N$ reads at most the terms $(y_h)_i$ of the argument y_h where $0 \leqslant i \leqslant \max\left(0, N - \mathrm{sh}(\Gamma, j, \bar{h})\right)$.

Proof. By induction on the index j in the result sequence. When j corresponds to an input, *i.e.* $-(\ell-1)\leqslant j\leqslant 0$, the result c_j equals to the input $y_{j+(\ell-1)}$ so that the proposition is easily checked.

Now recursively for indices j corresponding to operations, *i.e.* $j>0$. If $\Gamma_j=(p^s\times\_;u)$, then for all $N\in\mathbb{N}$, $(c_j)_N=(p^s\,c_u)_N=(c_u)_{N-s}$ which, by assumption, reads at most the p-adic coefficients $(y_h)_i$ of the argument y_h where $0\leqslant i\leqslant\max(0,N-s-\mathrm{sh}(\Gamma,j,\bar{h}))$. So the definition matches.

If $\Gamma_j=(\cdot;u,v)$, then for all $N\in\mathbb{N}$, $(c_j)_N=(c_u\cdot c_v)_N$. Since the product $c_u\cdot c_v$ is done in Algorithm `OnlineEvaluation` by an on-line algorithm, the term $(c_j)_N$ depends only on the terms up to N of c_u and c_v, and the proposition follows.

The other cases can be treated similarly. □

Given any s.l.p. Γ, its shift index $\mathrm{sh}(\Gamma)$ can be computed automatically thanks to Definition 2.11. As an important consequence of Proposition 2.12, if an s.l.p. $\boldsymbol{\Psi}$ has a positive shift, then the computation of $(\boldsymbol{\Psi}(\boldsymbol{y}))_N$ does not read $\boldsymbol{y}_N$.

Example 2.13. We carry on with the notations of Warning 2.6. Recall that we remarked in Warning 2.6 that $(\Phi(y)_N)$ involved y_ℓ for $0\leqslant\ell\leqslant N$. We have now the tools to explain this. The shift of the s.l.p. Γ with one argument associated to the arithmetic expression $Z\mapsto Z^2+X$ (see Remark 2.2) satisfies

$$\begin{aligned}\mathrm{sh}(\Gamma) &= \min(\mathrm{sh}(Z\mapsto Z^2),\mathrm{sh}(Z\mapsto X))\\ &= \min(\min(\mathrm{sh}(Z\mapsto Z),\mathrm{sh}(Z\mapsto Z)),+\infty)\\ &= \min(\min(0,0),+\infty)\\ &= 0.\end{aligned}$$

Hence Proposition 2.12 gives that the computation of the ith term output of Φ: $Z\mapsto Z^2+X$ reads the jth term of the input with $0\leqslant j\leqslant i$, as observed in Warning 2.6.

Example 2.14. Here is a solution to the issue raised in Warning 2.6. Consider the s.l.p. deduced from the expression

$$\Psi\colon Z\mapsto X^2\times\left(\frac{Z}{X}\right)^2+X.$$

Then $\mathrm{sh}(\Psi)=1$, since

$$\begin{aligned}\mathrm{sh}(Z\mapsto X^2\times(Z/X)^2) &= \mathrm{sh}(Z\mapsto(Z/X)^2)+2\\ &= \mathrm{sh}(Z\mapsto Z/X)+2\\ &= \mathrm{sh}(Z\mapsto Z)+1.\end{aligned}$$

So Proposition 2.12 ensures that the s.l.p. Ψ solves the problem raised in Warning 2.6.

Still, we detail the first steps of the new algorithm to convince even the most skeptical reader. Again, let us specialize Algorithm `OnlineRecursivePadic` in our case. The divisions and multiplications by X induce directly a shift in the step of the relaxed multiplication.

Algorithm 2.2

Input: $N \in \mathbb{N}$
Output: $a \in R_p$

1. $a = 0\ (= y_0)$
2. $[c_1, ..., c_3] = [0, 0, 0]$
3. **for** i from 1 to N
   a. $c_1 = a/X$
   b. $c_2 = \texttt{RelaxedProductStep}(c_2, c_1, c_1, i-2)$
   c. $c_3 = X^2 \times c_2$
   d. $a = \texttt{LazyAddStep}(a, c_3, X, i)$
4. **return** a

At Step 0 on the example, we do

$$c_1 = a/X = 0,\ c_2 = c_2,\ c_3 = X^2 \times c_2 = 0,\ a = a + (c_3)_0 + 0 = 0.$$

Then at Step 1, the following computations are done

$$\begin{array}{cc} c_1 = a/X = 0, & c_2 = c_2, \\ c_3 = X^2 \times c_2 = 0, & a = a + ((c_3)_1 + 1)\, X = X. \end{array}$$

Step 2 computes

$$\begin{array}{cc} c_1 = a/X = 1, & c_2 = c_2 + ((c_1)_0)^2 = 1, \\ c_3 = X^2 \times c_2 = X^2, & a = a + ((c_3)_2 + 0)\, X^2 = X + X^2. \end{array}$$

Step 3 computes

$$\begin{array}{cc} c_1 = a/X = 1 + X, & c_2 = c_2 + 2\,(c_1)_0\,(c_1)_1\, X = 1 + 2\,X, \\ c_3 = X^2 \times c_2 = X^2 + 2\,X^3, & a = a + (c_3)_3\, X^3 = X + X^2 + 2\,X^3. \end{array}$$

Finally Step 4 computes

$$\begin{array}{c} c_1 = a/X = 1 + X + 2\,X, \\ c_2 = c_2 + (2\,(c_1)_0\,(c_1)_2 + ((c_1)_1 + (c_1)_2\, X)^2)\, X^2 = 1 + 2\,X + 5\,X^2 + 4\,X^3 + 4\,X^4, \\ c_3 = X^2 \times c_2 = X^2 + 2\,X^3 + 5\,X^4 + 4\,X^5 + 4\,X^6, \\ a = a + (c_3)_4\, X^4 = X + X^2 + 2\,X^3 + 5\,X^4 \end{array}$$

which is still correct. If you look at Step 4 in terms of coefficients of a, we see that the shift is 1 because we do not read a_4:

$$\begin{array}{c} c_1 = a/X = a_1 + a_2\, X + \cdots, \\ c_2 = a_1^2 + 2\,a_1\,a_2\, X + (2\,a_1\,a_3 + (a_2 + a_3\, X)^2)\, X^2, \\ c_3 = a_1^2\, X^2 + 2\,a_1\,a_2\, X^3 + (2\,a_1\,a_3 + (a_2 + a_3\, X)^2)\, X^4, \\ a = a + (2\,a_1\,a_3 + a_2^2)\, X^4. \end{array}$$

We use only the coefficients a_1, a_2, a_3 at Step 4, which coincide with y_1, y_2, y_3. Therefore no error is introduced, even in the anticipated computations. In a word, we have solved the dependency issue in this example.

We are now able to express which s.l.p.'s $\mathbf{\Psi}$ are suited to the implementation of recursive p-adic numbers.

Definition 2.15. *Let $\boldsymbol{y} \in (R_p)^\ell$ be a vector of p-adics and $\mathbf{\Psi}$ be an s.l.p. with ℓ inputs, ℓ outputs and operations in Ω'.*

Then, $\mathbf{\Psi}$ is said to be a shifted algorithm *that compute $\boldsymbol{y}$ if*

- $\mathrm{sh}(\mathbf{\Psi}) \geqslant 1$,
- *$\mathbf{\Psi}$ is executable on $\boldsymbol{y}$ over the R-algebra R_p.*

A shifted algorithm is a recursive operator, but with tighter conditions.

Proposition 2.16. *If $\mathbf{\Psi}$ is a shifted algorithm that computes $\boldsymbol{y}$ then $\boldsymbol{y}$ are recursive p-adics and $\mathbf{\Psi}$ is a recursive operator that allows the computation of $\boldsymbol{y}$.*

Proof. We prove that the output of the on-line algorithm $\mathbf{\Psi}^n = \mathbf{\Psi} \circ \cdots \circ \mathbf{\Psi}$ on the input $\boldsymbol{y}_0$ coincides with $\boldsymbol{y}$ at precision $n+1$. This result is true for $n=0$. We prove it recursively on n.

Assume the claim is verified for n and let us prove it for $n+1$. If we denote by $\boldsymbol{y}_{(n)} := \mathbf{\Psi}^n(\boldsymbol{y}_0)$, we know that $\nu_p(\boldsymbol{y} - \boldsymbol{y}_{(n)}) \geqslant n+1$. Now in the steps $0, \ldots, n+1$ of the on-line computation of $\mathbf{\Psi}(\boldsymbol{y}_{(n)})$, only the p-adic coefficients of $\boldsymbol{y}_{(n)}$ in p^i are read for $i \leqslant n$ because $\mathrm{sh}(\mathbf{\Psi}) \geqslant 1$. So one has the following equalities between p-adic coefficients

$$(\boldsymbol{y}_{(n+1)})_i = (\mathbf{\Psi}(\boldsymbol{y}_{(n)}))_i = (\mathbf{\Psi}(\boldsymbol{y}))_i = \boldsymbol{y}_i$$

for $i \leqslant n$ and finally $\nu_p(\boldsymbol{y} - \boldsymbol{y}_{(n+1)}) \geqslant n+2$. □

Next proposition is the cornerstone of complexity estimates regarding recursive p-adics. We denote by $\mathsf{R}(N)$ the cost of multiplying two elements of R_p at precision N by an on-line algorithm (see Chapter 1).

Proposition 2.17. *Let $\mathbf{\Psi}$ be a shifted algorithm for recursive p-adics $\boldsymbol{y}$ whose length is L and multiplicative complexity is L^*. Then, the vector of p-adics $\boldsymbol{y}$ can be computed at precision N in time $L^*\,\mathsf{R}(N) + \mathcal{O}(L\,N)$.*

Proof. We use Algorithm `OnlineRecursivePadic` to compute $\boldsymbol{y}$. We have to prove that this algorithm is correct if $\mathbf{\Psi}$ is a shifted algorithm.

For this matter it is sufficient to prove that in the loop of Algorithm `OnlineRecursivePadic`, the correct p-adic coefficients of $\boldsymbol{y}$ are written in $\boldsymbol{a}$ before they are read by a call to `OnlineEvaluationStep`.

Since $\mathrm{sh}(\mathbf{\Psi}) \geqslant 1$, Proposition 2.12 tells us that the Nth p-adic coefficients of $\boldsymbol{a}$ are not read before step $N+1$ of `OnlineEvaluationStep`. At step 0 of the loop of Algorithm `OnlineRecursivePadic`, the p-adic coefficient $\boldsymbol{a}_0$ equals to $\boldsymbol{y}_0$. Therefore the computations of `OnlineEvaluationStep`$(\mathbf{\Psi}, \boldsymbol{a}, [c_1, \ldots, c_L], 0)$ are correct, *i.e.* they are the same than if $\boldsymbol{y}$ was given in input instead of $\boldsymbol{a}$.

At step 1, the call to `OnlineEvaluationStep`$(\mathbf{\Psi}, \boldsymbol{a}, [c_1, \ldots, c_L], 1)$ will only read $\boldsymbol{a}_0$ and carry correct computations, giving $\boldsymbol{y}_1 = (\mathbf{\Psi}(\boldsymbol{a}))_1$. At step 2, the call to `OnlineEvaluationStep`$(\mathbf{\Psi}, \boldsymbol{a}, [c_1, \ldots, c_L], 2)$ is known to read at most $\boldsymbol{a}_0, \boldsymbol{a}_1$, which coincide with $\boldsymbol{y}_0, \boldsymbol{y}_1$. So we will have $\boldsymbol{y}_2 = (\mathbf{\Psi}(\boldsymbol{a}))_2$. And so on.

The key point of our demonstration is that at each step, since the p-adic coefficients of $\boldsymbol{a}$ which are read in the call to `OnlineEvaluationStep` coincides with the ones of $\boldsymbol{y}$, Algorithm `OnlineEvaluationStep` does the same computation as if $\boldsymbol{y}$ was given in input instead of $\boldsymbol{a}$, and so computes correctly $\boldsymbol{\Psi}(\boldsymbol{y})$.

Therefore the cost of the computation of $\boldsymbol{y}$ *is exactly the cost of the evaluation* of $\boldsymbol{\Psi}(\boldsymbol{y})$ in R_p. We recall that addition in $R_p \times R_p$, subtraction in $R_p \times R_p$ and multiplication in $R \times R_p$ (that is operations in R) up to the precision N can be computed in time $\mathcal{O}(N)$. Scalars from R are decomposed in R_p in constant complexity. Finally, multiplications in $R_p \times R_p$ are done in time $\mathsf{R}(N)$. Now the multiplicative complexity L^* of $\boldsymbol{\Psi}$ counts exactly the latter operation. □

Of course, if some multiplications in the evaluation of $\boldsymbol{\Psi}$ are between finite length p-adics, they cost less than $\mathsf{R}(N)$. An important special case concerns multiplications between a p-adic and another p-adic of length d, which can be done in time $\mathcal{O}(N\,\mathsf{R}(d)/d)$ instead of $\mathsf{R}(N)$.

Remark 2.18. The important property used in the proof of Proposition 2.17 is that Algorithm `OnlineEvaluation`($\boldsymbol{\Phi}$, _, N) has shift 1. We can extend the set of operations Ω' of our s.l.p.'s and adapt the rules of computation of $\mathrm{sh}(\boldsymbol{\Psi})$ consequently, Proposition 2.17 will remain correct as long as Algorithm `OnlineEvaluation`($\boldsymbol{\Phi}$, _, N) has shift 1.

Newton iteration Under the assumptions of Proposition 2.4, we can use the Newton iteration algorithm (also called Hensel lifting) to compute $\boldsymbol{y}$. Let us recall the mechanism of this lifting method.

If $\boldsymbol{f} := \mathrm{Id} - \boldsymbol{\Phi} \in R_p[Y_1, ..., Y_\ell]^\ell$, then $\boldsymbol{y}$ is a zero of the polynomials $\boldsymbol{f}$. Moreover since $\mathrm{Id} - \mathrm{Jac}_{\boldsymbol{f}}(\boldsymbol{y}_0) = \mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{y}_0)$ has positive valuation, the Jacobian matrix $\mathrm{Jac}_{\boldsymbol{f}}(\boldsymbol{y})$ is invertible over R_p. Then we define recursively $\boldsymbol{y}_{(0)} = \boldsymbol{y}_0$ and for all $N \in \mathbb{N}$

$$\boldsymbol{y}_{(N+1)} = \boldsymbol{y}_{(N)} - \mathrm{Jac}_{\boldsymbol{f}}(\boldsymbol{y}_{(N)})^{-1}\, \boldsymbol{f}(\boldsymbol{y}_{(N)}) \in (R_p)^\ell.$$

It can be shown that for all $N \in \mathbb{N}$, $\nu_p(\boldsymbol{y}_{(N)} - \boldsymbol{y}) \geqslant 2^N$ [GG03].

The Newton iteration algorithm, as well as our on-line lifting algorithm for recursive p-adics, applies to more general operators than polynomial function $\boldsymbol{\Phi}$ and $\boldsymbol{f}$. For example on power series, the operator $\boldsymbol{\Phi}$ can use differentiation and integration. The notion of shift and shifted algorithms can be extended to s.l.p.'s with these new operators.

Space complexity One drawback of the relaxed method for computing recursive p-adics is the space complexity. We have seen that we store the current state of each computation of $\boldsymbol{\Psi}$ in Algorithm `OnlineRecursivePadicStep`. This leads to a space complexity $\mathcal{O}(N\,L)$ to compute the recursive p-adic at precision N where L is the size of $\boldsymbol{\Psi}$.

The zealous approach to evaluate $\boldsymbol{\Psi}$ could use significantly less memory by freeing the result of a computation as soon as it is used for the last time. For this reason, zealous lifting based on Newton iteration should consume less memory.

Partie II

Lifting of linear equations

Chapitre 3

Linear algebra over p-adics

This chapter deals with the resolution of linear systems over the p-adics. Linear algebra problems are often classified into broad categories, depending on whether the matrix of the system is dense, sparse, structured, ... In the context of solving over the p-adics, most previous algorithms rely on lifting techniques using either Dixon's / Moenck-Carter's algorithm, or Newton iteration, and can to some extent exploit the structure of the given matrix.

In this chapter, we introduce an algorithm based on the p-recursive framework of Chapter 2, which can in principle be applied to all above families of matrices. We will focus on two important cases, *dense* and *structured* matrices, and show how our algorithm can improve on existing techniques in these cases.

The relaxed linear system solver applied to dense matrices is a common work with J. Berthomieu, published as a part of [BL12]. The application to structured matrices is a joint work in progress with É. Schost.

3.1 Overview

Assumptions on the base ring Throughout this chapter, we continue using some notation and assumptions introduced in Chapter 1: R is our base ring (typically, $\mathbb{Z}$ or $\Bbbk[X]$), p is a non-zero element in R (typically, a prime in $\mathbb{Z}$ or $X \in \Bbbk[X]$) and R_p is the completion of R for the p-adic topology (so we get for instance the p-adic integers, or the power series ring $\Bbbk[[X]]$). In order to simplify some considerations below regarding the notion of rank of a matrix over a ring, we will make the following assumption in all this chapter: *both R and R_p are domains*; this is the case in the examples above.

As before, we fix a set M of representatives of $R/(p)$, which allows us to define the *length* $\lambda(a)$ of a non zero p-adic $a \in R_p$; recall that we make the assumption that the elements of $R \subset R_p$ have finite length. We generalize the length function to vectors or matrices of p-adics by setting $\lambda(A) := \max_{1 \leqslant i \leqslant r, 1 \leqslant j \leqslant s} (\lambda(A_{i,j}))$ if $A \in \mathcal{M}_{r \times s}(R_p)$.

Problem statement We consider a linear system of the form $A = B \cdot C$, where A and B are known, and C is the unknown. The matrix A belongs to $\mathcal{M}_{r\times s}(R_p)$ and $B \in \mathcal{M}_{r\times r}(R_p)$ is invertible; we solve the linear system $A = B \cdot C$ for $C \in \mathcal{M}_{r\times s}(R_p)$. We make the natural assumption that $s \leqslant r$; the most interesting cases are $s = 1$ (which amounts to linear system solving) and $s = r$, which contains in particular the problem of inverting B (our algorithm handles both cases in a uniform manner).

A major application of p-adic linear system solving is actually to solve systems over R (in the two contexts above, this means systems with integer, resp. polynomial coefficients), by means of lifting techniques (the paper [MC79] introduced this idea in the case of integer linear systems). In such cases, the solution C belongs to $\mathcal{M}_{r\times s}(Q)$, where Q is the fraction field of R, with a denominator invertible modulo p. Using p-adic techniques, we can compute the expansion of C in $\mathcal{M}_{r\times s}(R_p)$, from which C itself can be reconstructed by means of rational reconstruction — we will focus on the lifting step, and we will not detail the reconstruction step here.

In order to describe such situations quantitatively, we will use the following parameters: the length of the entries of A and B, that is, $d := \max(\lambda(A), \lambda(B))$, and the precision N to which we require C; thus, we will always be able to suppose that $d \leqslant N$. The case $N = d$ corresponds to the resolution of p-adic linear systems proper, whereas solving systems over R often requires to take a precision $N \gg d$. Indeed, in that case, we deduce from Cramer's formulas that the numerators and denominators of C have length $\mathcal{O}(r\,(d + \log(r)))$, so that we take N of order $\mathcal{O}(r\,(d + \log(r)))$ in order to make rational reconstruction possible.

For computations with structured matrices, we will use a different, non-trivial representation for B, by means of its "generators"; then, we will denote by d' the length of these generators. Details are given below.

Complexity model Throughout this chapter, we represent all p-adics through their base-M expansion, and we measure the cost of an algorithm by the number of arithmetic operations on p-adics of length 1 (*i.e.* with only a constant coefficient) it performs, as explained in Chapter 1.

The algorithms in this chapter will rely on the notion of *shifted decomposition*: a shifted decomposition of a p-adic $a \in R_p$ is simply a pair $(\sigma_a, \delta_a) \in R_p^2$ such that $a = \sigma_a + p\,\delta_a$. A simple particular case is $(a \bmod p, a \text{ quo } p)$; this is by no means the only choice. This notion carries over to matrices without difficulty.

We denote by $\mathsf{I}(N)$ the cost of multiplication of two p-adics at precision N and we let $\mathsf{R}(N)$ be the cost of multiplying two p-adics at precision N by an on-line algorithm. As in Chapter 1, we let further $\mathsf{M}(d)$ denote the arithmetic complexity of multiplication of polynomials of degree at most d over any ring (we will need this operation for the multiplication of structured matrices). Remark that when $R = \Bbbk[X]$, I and M are the same thing, but this may not be the case anymore over other rings, such as $\mathbb{Z}$.

Let next $\mathsf{I}(r, d)$ be the cost of multiplying two polynomials in $R_p[Y]$ with degree at most r and coefficients of length at most d. Since the coefficients of the product polynomial have length at most $2\,d + \lceil \log_2(r) \rceil$, we deduce that we can take

$$\mathsf{I}(r, d) = \mathcal{O}(\mathsf{M}(r)\ \mathsf{I}(d + \log(r)))$$

by working modulo p to the power the required precision; over $R_p = \Bbbk[[X]]$, the $\log(r)$ term vanishes since no carry occurs.

Let us focus on the corresponding on-line algorithm. We consider these polynomials as p-adics of polynomials, *i.e.* p-adic whose coefficients are polynomials in M. We denote by $\mathsf{R}(r, N)$ the cost of an on-line multiplication at precision N of polynomials of degrees at most r. As in Chapter 1, this cost is bounded by $\mathsf{R}(r, N) = \mathcal{O}(\mathsf{I}(r, N)\log(N))$ in the case of power series rings or p-adic integers. If the length d' of the coefficients of one operand is less than N, the cost reduces to $\mathcal{O}(N\,\mathsf{R}(r, d')/d')$.

Now, let us turn to matrix arithmetic. We let ω be such that we can multiply $r \times r$ matrices within $\mathcal{O}(r^\omega)$ ring operations over any ring. The best known bound on ω is $\omega \leqslant 2.3727$ [CW90, Sto10, VW11]. It is known that, if the base ring is a field, we can invert any invertible matrix in time $\mathcal{O}(r^\omega)$ base field operations. We will further denote by $\mathsf{MM}(r, s, d)$ the cost of multiplication of matrices A, B of sizes $(r \times r)$ by $(r \times s)$ over R_p, for inputs of length at most d. In our case $s \leqslant r$, and taking into account the growth of the length in the output, we obtain that $\mathsf{MM}(r, s, d)$ satisfies

$$\mathsf{MM}(r, s, d) = \mathcal{O}(r^2\, s^{\omega-2}\, \mathsf{I}(d + \log(r))),$$

since $\lambda(A \cdot B) \leqslant 2\,d + \lceil \log_2(r) \rceil$; the exponents on r and s are obtained by partitioning A and B into square blocks of size s.

Let us now consider the relaxed product of p-adic matrices, *i.e.* p-adic whose coefficients are matrices over M. We denote by $\mathsf{MMR}(r,\ s,\ N)$ the cost of the relaxed multiplication of a p-adic matrix of size $r \times r$ by a p-adic matrix of size $r \times s$ at precision N. As in Chapter 1, we can connect the cost of off-line and on-line multiplication algorithms by

$$\mathsf{MMR}(r, s, N) = \mathcal{O}(\mathsf{MM}(r, s, N)\log(N))$$

in the case of power series rings or p-adic integers. Likewise, we also notice that the relaxed multiplication of two matrices $A, B \in (\mathcal{M}_{r\times s}(R))_{(p)}$ at precision N with $d := \lambda(A) \leqslant N$ takes time $\mathcal{O}(N\,\mathsf{MMR}(r, s, d)/d)$.

Previous work The first algorithm we will mention is due to Dixon [Dix82]; it finds one p-adic coefficient of the solution C at a time and then updates the matrix A. On the other side of the spectrum, one finds Newton's iteration, which doubles the precision of the solution at each step (and can thus benefit from fast p-adic multiplication); however, this algorithm computes the whole inverse of B at precision N, which can be too costly when we only want one vector solution.

Moenck-Carter's algorithm [MC79] is a variant of Dixon's algorithm that works with p^ℓ-adics instead of p-adics. It takes advantages of fast truncated p-adic multiplication but requires that we compute the inverse of B at precision d (for which Newton iteration is used).

Finally, Storjohann's high-order lifting algorithm [Sto03] can be seen as a fast version of Moenck-Carter's algorithm, well-suited to cases where $d \ll N$. That algorithm was presented for $R = \Bbbk[X]$ and the result was extended to the integer case in [Sto05]. We believe that the result could carry over to any p-adic ring.

Historically, these algorithms were all introduced for dense matrices; however, most of them can be adapted to work with structured matrices. The exception is Storjohann's high-order lifting, which does not seem to carry over in a straightforward manner.

Main results The core of this chapter is an algorithm to solve linear systems by means of relaxed techniques; it is obtained by proving that the entries of the solution $C = B^{-1} \cdot A$ are p-recursive. In other words, we show that C is a fixed point for a suitable shifted operator.

This principle can be put to use for several families of matrices; we detail it for dense and structured matrices. Taking for instance $s = 1$, to compute C at precision N, the cost of the resulting algorithm will (roughly speaking) involve the following:

- the inversion of B modulo (p),
- $\mathcal{O}(N)$ matrix-vector products using the inverse of B modulo (p), with a right-hand side vector whose entries have length 1,
- $\mathcal{O}(1)$ matrix-vector product using B, with a right-hand side vector whose entries are relaxed p-adics.

Tables 3.1 and 3.2 give the resulting running time for the case of dense matrices, together with the results based on previous algorithms mentioned above; recall that $d = \lambda(B)$ and that N is the target precision. In the first table, we are in the general case $1 \leqslant s \leqslant r$; in the second one, we take $R = \Bbbk[X]$ and $s = 1$, and we choose two practically meaningful values for N, respectively $N = d$ and $N = r\,d$ (which was mentioned above). For the high-order lifting, the * indicates that the result is formally proved only for $R_p = \Bbbk[[X]]$ and $R = \mathbb{Z}$. The complexity $\mathsf{MM}(r, s\,N/d, 1)$ that appears in this case is bounded by $\mathsf{MM}(r, s\,N/d, 1) = r^{\omega-1}\,s\,N/d$.

Most previous complexity results are present in the literature, so we will not reprove them all; we only do it in cases where small difficulties may arise. For instance, Newton's algorithm and its cost analysis extend in a straightforward manner, since we only do computations modulo powers of p, which behave over general p-adics as they do over e.g. $R_p = \Bbbk[[X]]$; thus, we will not reprove the running time in this case. On the other hand, we will re-derive the cost of Dixon's and Moenck-Carter's algorithms, since they involve computations in R_p itself (*i.e.*, without reduction modulo a power of p), and considerations about the lengths of the operands play a role.

In most entries (especially in the first table), two components appear: the first one involves inverting the matrix B modulo (p), or a higher power of p and is independent of N; the second one describes the lifting process itself. In some cases, the cost of the first step can be neglected compared to the cost of the second one.

It appears in the last table that for solving up to precision $N = d$, our algorithm is the fastest among the ones we compare; for $N = r\,d$, Storjohann's high-order lifting does best (as it is specially designed for such large precisions).

Algorithm	Cost
Dixon	$\mathcal{O}(r^\omega + \mathsf{MM}(r,s,1)\ N\ d)$
Moenck-Carter	$\mathcal{O}\big(r^\omega\, \mathsf{I}(d) + \mathsf{MM}(r,s,d)\ \frac{N}{d}\big)$
Newton iteration	$\mathcal{O}(r^\omega\ \mathsf{I}(N))$
High-order lifting*	$\mathcal{O}\big(r^\omega \log(\frac{N}{d})\ \mathsf{I}(d) + \mathsf{MM}(r, s\,\frac{N}{d}, 1)\ \mathsf{I}(d)\big)$
Our algorithm	$\mathcal{O}\Big(r^\omega + N\ \frac{\mathsf{MMR}(r,s,d)}{d}\Big)$

Table 3.1. Cost of solving $A = B \cdot C$ for dense matrices

Algorithm	$N = d$	$N = r\,d$
Dixon	$\tilde{\mathcal{O}}(r^\omega + r^2\,d^2)$	$\tilde{\mathcal{O}}(r^3\,d^2)$
Moenck-Carter	$\tilde{\mathcal{O}}(r^\omega\,d)$	$\tilde{\mathcal{O}}(r^3\,d)$
Newton iteration	$\tilde{\mathcal{O}}(r^\omega\,d)$	$\tilde{\mathcal{O}}(r^{\omega+1}\,d)$
High-order lifting*	$\tilde{\mathcal{O}}(r^\omega\,d)$	$\tilde{\mathcal{O}}(r^\omega\,d)$
Our algorithm	$\tilde{\mathcal{O}}(r^\omega + r^2\,d)$	$\tilde{\mathcal{O}}(r^3\,d)$

Table 3.2. Simplified cost of solving $A = B \cdot C$ for dense matrices over $R_p = \mathbb{k}[[X]]$, with $s = 1$

Next, we discuss the situation for structured matrices; for that, a brief reminder is in order (for a thorough presentation, see [Pan01]).

A typical family of structured matrices are Toeplitz matrices, which are invariant along diagonals; exploiting this structure, one can multiply and invert such matrices in quasi-linear time. In this chapter, we will consider structured matrices as being matrices which are "close" to being Toeplitz. Formally, let us define the operator

$$\begin{array}{rccc} \phi_+\colon & \mathcal{M}_{r\times r}(R_p) & \to & \mathcal{M}_{r\times r}(R_p) \\ & A & \mapsto & A - A', \end{array}$$

where A' is obtained by shifting A down and right by one unit. If A is Toeplitz, $\phi_+(A)$ is zero, except in the first row and column; the key remark is that in this case, $\phi_+(A)$ has a small rank (at most 2), and can be written $\phi_+(A) = G \cdot H^t$, with G and H matrices of sizes $r \times 2$, with entries in R_p.

The key idea is then to measure the "structure" of the matrix A as the rank of $\phi_+(A)$, which is called its *displacement rank*, usually denoted by $\alpha(A)$. If $\alpha(A) \leqslant \alpha$, then there exist matrices G and H in $\mathcal{M}_{r\times\alpha}(R_p)$ such that $\phi_+(A) = G \cdot H^t$.

The key idea of algorithms for structured matrices is to use such generators as a compact data structure to represent A, since they can encode A using $\mathcal{O}(\alpha\ r)$ elements of R_p instead of r^2. As typical examples, note that the displacement rank

of a Sylvester matrix is at most 2; more generally, matrices coming from e.g. Padé-Hermite approximation problems have small displacement ranks. The last important property of structured matrices is that the matrix-vector multiplication $A \cdot V$ for $A \in \mathcal{M}_{r\times r}(R_p)$ and $V \in \mathcal{M}_{r\times 1}(R_p)$ boils down to polynomial multiplication, so that it costs $\mathcal{O}(\alpha\,\mathsf{M}(r))$ (we will also need to pay attention to the precision of the arguments).

Table 3.3 recalls previously known results about solving structured linear systems and shows the running time of our algorithm. Recall that here, d' denotes the length of the generators of B and N is still the target precision. As before, Table 3.4 gives simplified results for $s=1$ and $N=d'$ and $N=r\,d'$.

Previous algorithms can all be found in [Pan01] for rings such as $R=\mathbb{Z}$ and $R=\Bbbk[X]$, so as in the case of dense matrices, we will only prove those cost estimates where attention must be paid to issues such as the length of the p-adics. Note that the high-order lifting entry has disappeared, since we do not know how to extend it to the structured case. As in the dense case, the running times involve two components: inverting the matrix modulo (p), then the lifting itself.

Algorithm	Cost
Dixon	$\mathcal{O}(\alpha^2\,\mathsf{M}(r)\log(r)+\alpha\,s\,\mathsf{M}(r)\,N\,d')$
Moenck-Carter	$\mathcal{O}\Big(\alpha^2\,\mathsf{M}(r)\log(r)+\alpha^2\,\mathsf{M}(r)\,\mathsf{I}(d')+\alpha\,s\,N\,\frac{\mathsf{I}(r,\,d')}{d'}\Big)$
Newton iteration	$\mathcal{O}(\alpha^2\,\mathsf{M}(r)\log(r)+\alpha^2\,\mathsf{M}(r)\,\mathsf{I}(N)+\alpha\,s\,\mathsf{M}(r)\,\mathsf{I}(N))$
Our algorithm	$\mathcal{O}\Big(\alpha^2\,\mathsf{M}(r)\log(r)+\alpha\,s\,N\,\frac{\mathsf{R}(r,\,d')}{d'}\Big)$

Table 3.3. Cost of solving $A=B\cdot C$ for structured matrices

Algorithm	$N=d'$	$N=r\,d'$
Dixon	$\tilde{\mathcal{O}}\big(\alpha^2\,r+\alpha\,r\,d'^2\big)$	$\tilde{\mathcal{O}}\big(\alpha\,r^2\,d'^2\big)$
Moenck-Carter	$\tilde{\mathcal{O}}(\alpha^2\,r\,d')$	$\tilde{\mathcal{O}}(\alpha\,r^2\,d')$
Newton iteration	$\tilde{\mathcal{O}}(\alpha^2\,r\,d')$	$\tilde{\mathcal{O}}(\alpha^2\,r^2\,d')$
Our algorithm	$\tilde{\mathcal{O}}(\alpha^2\,r+\alpha\,r\,d')$	$\tilde{\mathcal{O}}(\alpha\,r^2\,d')$

Table 3.4. Simplified cost of solving $A=B\cdot C$ for structured matrices over $\Bbbk[[X]]$, with $s=1$

To summarize, in all these cases, our algorithm performs at least as well, and often better, than previous algorithms.

The relaxed linear system solver applied to dense matrices was published as a part of [BL12]. It has been implemented inside the MATHEMAGIX computer algebra system [HLM+02]. The application to structured matrices is a joint work in progress with É. SCHOST.

3.2 Structured matrices

While we need only fairly standard results about dense matrices, we believe it is worth recalling a few facts about the structured case. We will need very little of the existing results on structured matrices: mainly, how to reconstruct a matrix from its generators, as well as a few properties of the inverse of a structured matrix.

Let us start with a discussion of the inverse of A (assuming A is invertible). Then, is it know that the displacement rank $\alpha(A^{-1})$ is at most $\alpha(A)+2$, so that A^{-1} can be represented in a compact manner. What's more, generators of A^{-1} can be computed in time $\mathcal{O}(\alpha^2\,\mathsf{M}(r)\log(r))$ (using a Las-Vegas algorithm); this is called the Morf / Bitmead-Anderson algorithm [Mor74, Mor80, BA80].

At the heart of most algorithms for structured matrices lies the following question. Let G, H be generators for a matrix A. To use the generators G and H as a data structure, we must be able to recover A from these matrices. Indeed, the operator ϕ_+ is bijective, and it can be inverted as follows. Denote by H_i and G_i the columns of G and H, for $1 \leqslant i \leqslant \alpha$. For any $V = (v_0, \ldots, v_{r-1}) \in R_p^r$, we define the lower (resp. upper) triangular matrix $L(V)$ (resp. $U(V)$) by

$$L(V) := \begin{pmatrix} v_0 & 0 & \cdots & 0 \\ v_1 & v_0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ v_{r-1} & \cdots & v_1 & v_0 \end{pmatrix} \in \mathcal{M}_{r\times r}(R_p)$$

and $U(V) := L(V)^t$. Then, for any matrices $G, H \in \mathcal{M}_{r\times\alpha}$, we have the equivalence

$$\phi_+(A) = G\cdot H^t \quad\Leftrightarrow\quad A = \sum_{i=1}^{\alpha} L(G_i)\cdot U(H_i).$$

This representation of A is essential to perform the matrix-vector multiplication by A efficiently. For $n \in \mathbb{N}$ and $P = \sum_{i=0}^{n} P_i\, Y^i \in R_p[Y]$, we denote by $\mathrm{rev}_n(P)$ the reverse polynomial of P defined by

$$\mathrm{rev}_n(P) := \sum_{i=0}^{n} P_{n-i}\, Y^i \in R_p[Y]_{\leqslant n}.$$

Besides, to a vector $V := [v_0, \ldots, v_{r-1}]^t \in \mathcal{M}_{r\times 1}(R_p)$, we associate the polynomial $v \in R_p[X]$ defined by $v := \sum_{i=0}^{r-1} v_i\, X^i$. This association is bijective. Next, if a, c, $v \in R_p[Y]_{<r}$ are the polynomials associated to some vectors $A, C, V \in \mathcal{M}_{r\times 1}(R)$, then

$$c = \begin{cases} \mathrm{rev}_{r-1}(a\,\mathrm{rev}_{r-1}(v)) & \text{if } C = U(A)\cdot V \\ a\,v \bmod Y^r & \text{if } C = L(A)\cdot V. \end{cases} \tag{3.1}$$

This shows that given generators for A of size $r\times\alpha$, the matrix-vector multiplications $A\cdot V$ can be done using $2\,\alpha$ short products of polynomials in degree r.

To estimate costs precisely, we have to take into account the size of the operands. We will thus consider the length d' of the entries of the displacement generators. Then, if a vector V has length d' as well, the matrix-vector multiplication $B\cdot V$ costs $\mathcal{O}(\alpha\,\mathsf{I}(r,d'))$, with $\alpha := \alpha(B)$. Indeed, the cost of all multiplications is easily seen to be controlled by the above bound; the cost of additions follows from Lemma 1.1.

We can further deduce an on-line algorithm for the matrix-vector multiplication $B \cdot V$. For the polynomial multiplication of $R_p[Y]$ of Equation (3.1), use on-line algorithms on p-adic of polynomials. This algorithm is on-line with respect to the entries of the displacement generators G, H of B and with respect to V. It computes $B \cdot V$ at precision N in time $\mathcal{O}(\alpha\, \mathsf{R}(r, N))$. If the length $d' := \max(\lambda(G), \lambda(H))$ of the displacement generators is less than N, then the on-line multiplication $B \cdot V$ takes time $\mathcal{O}(\alpha\, N\, \mathsf{R}(r, d')/d')$.

Since in many situations we will encounter, the matrix B is known, we can adapt the latter algorithm to be half-line, that is off-line in B and on-line in V. For this matter, just replace on-line multiplication algorithms in R_p by half-line algorithms (which are slightly cheaper, see Chapter 1).

3.3 Solving linear systems

In this section, we give the details of the algorithm underlying the new results in Tables 3.1 to 3.4. We start by recalling Dixon's algorithm (and briefly mention the closely related Moenck-Carter's algorithm). In the second subsection, we will show how this algorithm can be seen as a relaxed algorithm that computes C as a fixed point and is useful when B has small length; finally, the last subsection introduces the general algorithm.

As a preamble, note that any matrix $A \in \mathcal{M}_{r\times s}(R_p)$ can be seen as a p-adic matrix, *i.e.* a p-adic whose coefficients are matrices over M. In this case, the coefficient matrix of index n will be denoted by $A_n \in \mathcal{M}_{r\times s}(M)$, so that $A = \sum_{n=0}^{\infty} A_n\, p^n$. We will use this notation frequently.

In this section, we denote by $d := \max(\lambda(A), \lambda(B))$ the maximum length of entries of A and B. If we assume that B is structured, its displacement rank is $\alpha := \alpha(B)$; in that case, as input, we assume that we are given generators G, H for B with entries in R_p, and we let $d' := \max(\lambda(G), \lambda(H))$. In all cases, we want C at precision N, so we suppose that $d, d' \leqslant N$.

3.3.1 Dixon's and Moenck-Carter's algorithms

The paper [Dix82] presented a simple algorithm to solve an integer linear system via a p-adic lifting; we present here a straightforward extension to our slightly more general context. This algorithm is based on the following lemma.

Lemma 3.1. *Let $B \in \mathcal{M}_{r\times r}(R_p)$ invertible and $A, C \in \mathcal{M}_{r\times s}(R_p)$ such that $A = B \cdot C$. Then for all $i \in \mathbb{N}$, there exists $A^{(i)} \in \mathcal{M}_{r\times s}(R_p)$ such that*

$$C = B^{-1} \cdot A = C_0 + C_1\, p + \cdots + C_{i-1}\, p^{i-1} + p^i\, B^{-1} \cdot A^{(i)}. \tag{3.2}$$

Proof. One has $A - B \cdot (C_0 + C_1\, p + \cdots + C_{i-1}\, p^{i-1}) = A - B \cdot C = 0$ in $\mathcal{M}_{r\times s}(R/(p^i))$. So we can define $A^{(i)} := p^{-i}\,[A - B \cdot (C_0 + C_1\, p + \cdots + C_{i-1}\, p^{i-1})]$ in $\mathcal{M}_{r\times s}(R_p)$ that satisfies Equation (3.2). □

The algorithm follows: at each step in the **for** loop, the matrix A is updated; the proof of the previous lemma shows that we are precisely computing the sequence $A^{(i)}$.

In order to analyze the algorithm, we need the following lemma describing the cost of polynomial or matrix multiplication with p-adic coefficients, in cases where the operands have unbalanced lengths. We will need the following extension of Lemma 1.1.

Lemma 3.2. *The following holds:*

- *Let P be in $\mathcal{M}_{r\times r}(R_p)$ and Q in $\mathcal{M}_{r\times s}(R_p)$, with $\lambda(P) = d$ and $\lambda(Q) = 1$. Then we can compute $S = P\,Q$ in time $\mathcal{O}(\mathsf{MM}(r,s,1)\,d)$.*
- *Let P, Q be in $R_p[X]_{<r}$ with $\lambda(P) = d$ and $\lambda(Q) = 1$. Then we can compute $S = P\,Q$ in time $\mathcal{O}(\mathsf{I}(r,1)\,d)$.*

Proof. In both cases, the strategy is the same. If the p-adic decomposition of Q is $\sum_{i=0}^{d-1} Q_i\,p^i$, then $S = \sum_{i=0}^{d-1} (PQ_i)\,p^i$. This amounts to d multiplications between operands of length 1 and some additions. Since the length of the entries of $P\,Q_i$ are bounded by $\lceil \log_2(r) \rceil$, Lemma 1.1 bounds the cost of the final addition by $\mathcal{O}(d \log(r))$, which is negligible compared to the cost of multiplications. □

Algorithm - `Dixon`
Input: $A \in \mathcal{M}_{r\times s}(R_p)$, $B \in \mathcal{M}_{r\times r}(R_p)$ and $N \in \mathbb{N}$ **Output:** $C \in \mathcal{M}_{r\times s}(R_p)$ such that $A = B \cdot C \bmod p^N$
1. $\Gamma = B^{-1} \bmod p$
2. $C_0 := (\Gamma \cdot A) \bmod p$
3. **for** i from 1 to $N-1$
a. $A := (A - B \cdot C_{i-1}) \operatorname{quo} p$
b. $C_i := (\Gamma \cdot A) \bmod p$
4. **return** $C := \sum_{i=0}^{N-1} C_i\,p^i$

Proposition 3.3. *Algorithm* `Dixon` *is correct and its cost is summed up in the table*

Dense matrices	$\mathcal{O}(r^\omega + \mathsf{MM}(r,s,1)\,N\,d)$
Structured matrices	$\mathcal{O}(\alpha^2\,\mathsf{M}(r)\log(r) + \alpha\,s\,\mathsf{M}(r)\,N d')$

Table 3.5. *Cost of Dixon's algorithm depending on matrix representation*

Proof. We refer to [Dix82] for the proof of correctness of the algorithm (which readily follows from the previous lemma).

After computing Γ, at each step, the algorithm performs one multiplication $B \cdot C_{i-1}$ and one multiplication $\Gamma \cdot A$ modulo p, plus some additions, remainders and quotients whose cost is dominated by the multiplications.

Let us first study the cost for dense matrices. Computing Γ takes $\mathcal{O}(r^\omega)$ operations (since this is arithmetic modulo (p)). To compute $B \cdot C_{i-1}$, we apply the previous lemma, since B has length d and C_{i-1} has length 1; the cost is $\mathcal{O}(\mathsf{MM}(r, s, 1)\, d)$ using Lemma 3.2. Computing $\Gamma \cdot A \bmod p$ is cheaper, since we do all operations modulo p. Taking all i into account, we get the claimed result.

For structured matrices, notice that $B \bmod p$ is a structured matrix of rank at most α and so $\Gamma = B^{-1} \bmod p$ has rank $\alpha(\Gamma) \leqslant \alpha + 2$. Therefore the cost for structured matrices is $\mathcal{O}(\alpha^2\, \mathsf{M}(r) \log(r))$, for the computation of Γ, plus, for each i, the cost induced by the products $B \cdot C_{i-1}$ and $\Gamma \cdot A \bmod p$. The latter is negligible. The former is done by means of $\mathcal{O}(\alpha\ s)$ polynomial multiplications in degree r, with coefficients of lengths respectively d' and 1. The cost estimate follows from the previous lemma. □

Moenck-Carter's algorithm can be seen as a p^ℓ-adic variant of Dixon's, where we compute ℓ p-adic coefficients of C at a time (thus, the algorithm is formally the same, up to replacing p by p^ℓ). By suitably choosing ℓ, it allows us to benefit from fast p-adics multiplication algorithms.

One quickly sees that the optimal asymptotic cost in N is obtained by choosing $\ell = d$ (in the dense case) and $\ell = d'$ (in the structured case). This gives the costs reported in Table 3.6 below, which we justify now.

Dense matrices	$\mathcal{O}\big(r^\omega\, \mathsf{I}(d) + \mathsf{MM}(r, s, d)\, \frac{N}{d}\big)$
Structured matrices	$\mathcal{O}\big(\alpha^2\, \mathsf{M}(r) \log(r) + \alpha^2\, \mathsf{M}(r)\, \mathsf{I}(d') + \alpha\, s\, \mathsf{M}(r, d')\, \frac{N}{d'}\big)$

Table 3.6. Cost of Moenck-Carter's algorithm depending on matrix representation

The algorithm starts by computing the inverse Γ of B modulo p^ℓ; for this, we use Newton iteration, whose cost was recalled in the previous section. At each step of the loop, the algorithm computes $\Gamma \cdot A$ modulo p^ℓ and $B \cdot C_i$, where now C_i has length ℓ.

For dense matrices, taking $\ell = d$, the product $\Gamma \cdot A$ modulo p^ℓ takes time $\mathcal{O}(r^2\, s^{\omega-2}\, \mathsf{I}(d))$, which is always less than the cost of $B \cdot C_i$, that is $\mathcal{O}(\mathsf{MM}(r, s, d))$. Since the loop now has length N/d, it sums to the announced cost.

For structured matrices, we take $\ell = d'$. Using Newton iteration., the first inversion costs $\mathcal{O}(\alpha^2\, \mathsf{M}(r) \log(r) + \alpha^2\, \mathsf{M}(r)\, \mathsf{I}(d'))$. For each i in the main loop, the product $\Gamma \cdot A$ modulo p^ℓ takes time $\mathcal{O}(\alpha\, s\, \mathsf{M}(r)\, \mathsf{I}(d'))$, which is always less than the cost of computing $B \cdot C_i$, that is $\mathcal{O}(\alpha\, s\, \mathsf{M}(r, d'))$. Since we now do N/d' passes through the loop, this gives the announced cost.

3.3.2 The on-line point of view on Dixon's algorithm

We published in [BL12, Section 4.1] an on-line algorithm to solve linear systems $B \cdot C = A$ for C, well-adapted to cases where $\lambda(B)$ is small. With hindsight, we realized that this algorithm coincides with Dixon's algorithm.

In this subsection, we make this remark more precise: we prove that Dixon's algorithm is on-line, by presenting it slightly differently to write it as a fixed point algorithm `OnlineDixon`. Then we prove that this algorithm is a shifted algorithm. This will be useful for the next subsection, where we deal with cases where $\lambda(B)$ is arbitrary.

We will use two operators `Mul_rem` and `Mul_quo`, defined for $B \in \mathcal{M}_{r\times r}(R_p)$ and $A \in \mathcal{M}_{r\times s}(R_p)$ by

$$\begin{aligned} \texttt{Mul\_rem}(B,A) &:= \sum_{n\in\mathbb{N}} (B\cdot A_n \bmod p)\, p^n \in \mathcal{M}_{r\times s}(R_p) \\ \texttt{Mul\_quo}(B,A) &:= \sum_{n\in\mathbb{N}} (B\cdot A_n \operatorname{quo} p)\, p^n \in \mathcal{M}_{r\times s}(R_p) \end{aligned}$$

so that we have

$$B\cdot A = \texttt{Mul\_rem}(B,A) + p\,\texttt{Mul\_quo}(B,A).$$

We see on these definitions that `Mul_rem`$(B,\ A)$ and `Mul_quo`$(B,\ A)$ are on-line algorithms with respect to the input A.

Proposition 3.4. *Let $A \in \mathcal{M}_{r\times s}(R_p)$ and $B \in \mathcal{M}_{r\times r}(R_p)$. Suppose B is invertible modulo p and define $\Gamma := B^{-1} \bmod p$. Set $C_0 := (\Gamma\cdot A) \bmod p$ and let* `OnlineDixon` *denote the algorithm*

$$\texttt{OnlineDixon}(A,B,Y) := \texttt{Mul\_rem}(\Gamma, A - p\times \texttt{Mul\_quo}(B,Y)).$$

Then, Algorithm `OnlineDixon` *has shift 1 with respect to its input Y and has shift 0 with respect to its input A. Moreover,*

$$C = \texttt{OnlineDixon}(A,B,C).$$

Proof. First,

$$\begin{aligned} & & A &= \texttt{Mul\_rem}(B,C) + p\,\texttt{Mul\_quo}(B,C) \\ \Rightarrow & & \texttt{Mul\_rem}(B,C) &= A - p\,\texttt{Mul\_quo}(B,C) \\ \Rightarrow & & C &= \texttt{Mul\_rem}(\Gamma, A - p\,\texttt{Mul\_quo}(B,C)) \end{aligned}$$

so that $\Psi(C) = C$.

Next, for any $n \in \mathbb{N}$, the p-adic coefficient $\texttt{OnlineDixon}(A,B,C)_n$ requires the nth p-adic coefficient of $A - p\times \texttt{Mul\_quo}(B,C)$. This computation reads at most the coefficients A_i with $0 \leqslant i \leqslant n$, so $0 \in \mathcal{S}(\texttt{OnlineDixon}, 1)$ (see Definition 2.8). It also requires the coefficient $\texttt{Mul\_quo}(B,C)_{n-1}$, which reads at most the coefficients C_i with $0 \leqslant i \leqslant n-1$, so $1 \in \mathcal{S}(\texttt{OnlineDixon}, 3)$. □

Dixon's algorithm coincides with Algorithm `OnlineDixon`. Indeed, in the on-line algorithm presented here, the computation $A - p\ \texttt{Mul\_quo}(B,\ C)$ subtracts $\texttt{quo}(B\cdot C_i, p)$ to A at step i, which corresponds to the instruction

$$A := (A - B\cdot C_i) \operatorname{quo} p$$

in Dixon's algorithm. Similarly, the `Mul_rem` operations corresponds to the computation modulo p in Dixon's algorithm.

3.3.3 On-line solving of p-adic linear systems

For dense matrices, the running time analysis showed that Dixon's algorithm is satisfactory when $\lambda(B) = 1$, but not anymore when $\lambda(B)$ is large (since when $\lambda(B) \simeq N$, the behavior is then quadratic in N). The same holds for structured matrices — in that case, it is the length of the given generators that matters.

To by-pass this issue, we give our main result concerning the resolution of linear systems, which shows that the solution C of the system is a fixed point for an operator easily deduced from the original system, and whose matrix has better length properties than B. This is an extension of the algorithm for division of p-adics of [Hoe02, BHL11].

Proposition 3.5. *Let $A \in \mathcal{M}_{r\times s}(R_p)$ and $B \in \mathcal{M}_{r\times r}(R_p)$ be two matrices such that B_0 is invertible of inverse $\Gamma = B_0^{-1} \bmod p$. Let $C := B^{-1} \cdot A$ and $C_0 := (\Gamma \cdot A) \bmod p$. Let finally $(\sigma_B, \delta_B) \in \mathcal{M}_{r\times r}(R_p)^2$ be any shifted decomposition of B. We note $\texttt{OnlineSolve}(A, B, Y)$ the algorithm defined by*

$$\texttt{OnlineSolve}(A, B, Y) := \texttt{OnlineDixon}(A - p \times (\delta_B \cdot Y), \sigma_B, Y).$$

Then, the s.l.p. Ψ with operations in $\{+, -, \cdot, p^s \times \_, \_/p^s, \texttt{Mul\_rem}, \texttt{Mul\_quo}\} \cup R \cup R^c$ defined by

$$\Psi\colon \quad Y \longmapsto \texttt{OnlineSolve}(A, B, Y) \tag{3.3}$$

verifies that $\texttt{OnlineEvaluation}(\Psi, \_, N)$ has shift 1 and that $C = \Psi(C)$.

Proof. We begin by noticing that

$$\sigma_B \cdot C = (A - p \times (\delta_B \cdot C))$$

which gives $\Psi(C) = \texttt{OnlineDixon}(\sigma_B \cdot C, \sigma_B, C) = C$.

Next, we compute the shift of Ψ. Recall that Proposition 3.4 states that Algorithm `OnlineDixon` is on-line with respect to its first argument and has a shift 1 with respect to its third input. As a consequence, for any $n \in \mathbb{N}$, the computation of $\Psi(y)_n$ reads at most the p-adic coefficients $[A - p \times (\delta_B \cdot Y)]_i$ and Y_j with $0 \leqslant i \leqslant n$ and $0 \leqslant j \leqslant n-1$. Since $[A - p \times (\delta_B \cdot Y)]_i$ reads at most Y_k for $0 \leqslant k \leqslant n-1$, we have proved our assertion. □

The following proposition analyze the complexity in our two cases of interest. Let us recall that $\mathsf{R}(d)$ denotes the cost of the relaxed multiplication at precision N.

Proposition 3.6. *Let $B \in \mathcal{M}_{r\times r}(R_p)$ and $A \in \mathcal{M}_{r\times s}(R_p)$ be two p-adic matrices and note $d := \lambda(B)$ the length of B. Let $\alpha := \alpha(B)$ be the displacement rank of B and $d' := \max(\lambda(G), \lambda(H))$ where G, H are generators for B. We solve the linear system $B \cdot C = A$ at precision N so we can always assume that $N \geqslant d$.*

Then the computation costs of $C = B^{-1} \cdot A$ are displayed in the following table, depending on the matrix representation of B.

Dense representation	$\mathcal{O}\Big(r^\omega + N\,\frac{\mathsf{MMR}(r,s,d)}{d}\Big)$
Structured matrices	$\mathcal{O}\Big(\alpha^2\,\mathsf{M}(r)\log(r) + \alpha\,s\,N\,\frac{\mathsf{R}(r,d')}{d'}\Big)$

Table 3.7. *Cost of solving linear system for finite length matrices*

Proof. Proposition 3.5 tells us that for any shifted decomposition (σ_B, δ_B) of B, the s.l.p.

$$\Psi\colon \quad Y \mapsto \texttt{OnlineDixon}(A - p \times (\delta_B \cdot Y), \sigma_B, Y)$$

satisfies the hypothesis of Proposition 2.17, taking into consideration Remark 2.18. Therefore, this s.l.p. can be used to compute $B^{-1} \cdot A$ at precision N in the time necessary to evaluate Ψ at y.

If B is a dense matrix, we take the shifted decomposition $(\sigma_B, \delta_B) := (\sigma(B), \delta(B))$ of B. Since the p-adic matrix δ_B has length lesser or equal to d, the multiplication $\delta_B \cdot Y$ costs $\mathcal{O}(N\,\mathsf{MMR}(r,s,d)/d)$. Using Proposition 3.3 with $\lambda(B_0) = 1$, we conclude that the cost of solving for dense matrices is

$$\mathcal{O}(N\,\mathsf{MMR}(r,s,d)/d) + \mathcal{O}(\mathsf{MM}(r,s,1)\,N) + \mathcal{O}(\mathsf{MM}(r,r,1)).$$

For structured matrices B, we write B as

$$\begin{aligned} B &= \sum_{i=0}^{\alpha} L(G_i) \cdot U(H_i) \\ &= \underbrace{\sum_{i=0}^{\alpha} L(\sigma(G_i)) \cdot U(\sigma(H_i))}_{\sigma_B} + \underbrace{p\Big(\sum_{i=0}^{\alpha} L(\delta(G_i)) \cdot U(H_i) + L(\sigma(G_i)) \cdot U(\delta(H_i))\Big)}_{\delta_B}. \end{aligned}$$

We take the (σ_B, δ_B) of previous equation as a shifted decomposition for B. The important point is that $\alpha(\sigma_B) \leqslant \alpha$ and $\alpha(\delta_B) \leqslant 2\,\alpha$. Moreover the displacement generators of σ_B are $\sigma(G)$, $\sigma(H)$, which have length 1. The matrix multiplication $\delta_B \cdot Y$ costs $\mathcal{O}(\alpha\ N\ \mathsf{R}(r,\ d')/d')$. The cost of applying `OnlineDixon` is given by Proposition 3.3. Summing up, the cost of solving for structured matrices is

$$\mathcal{O}(\alpha\,s\,N\,\mathsf{R}(r,d')/d') + \mathcal{O}(\alpha\,s\,\mathsf{M}(r)\,N) + \mathcal{O}(\alpha^2\,\mathsf{M}(r)\log(r)). \qquad \square$$

Remark 3.7. For matrices B with length greater than 1, one should use Algorithm `OnlineSolve` instead of `OnlineDixon` as it is faster. However, we had to present Algorithm `OnlineDixon` for any matrices B of finite length. Indeed, in the latter proof in the structured matrix case, the matrix σ_B has length $\lceil \log_2(r) \rceil$.

3.4 Implementation and Timings

In this section, we display computation times in milliseconds for the univariate polynomial root lifting and for the computation of the product of the inverse of a matrix with a vector or with another square matrix. Timings are measured using one core of an INTEL XEON X5650 at 2.67 GHz running LINUX, GMP 5.0.2 [G+91] and setting $p = 536871001$ a 30 bit prime number.

Our implementation is available in the files whose names begin with `series_carry` or `p_adic` in the C++ library ALGEBRAMIX of MATHEMAGIX.

In the following tables, the first line, "Newton" corresponds to the classical Newton iteration [GG03, Algorithm 9.2] used in the zealous model. The second line "Relaxed" corresponds to our best variant. The last line gives a few details about which variant is used. We make use of the naive variant "N" and the relaxed variant "R". These variants differ only by the on-line multiplication algorithm used in Algorithm `OnlineEvaluationStep` inside Algorithm `OnlineRecursivePadic` to compute the recursive p-adics (see Section 2.2.2). The naive variant calls Algorithm `LazyMulStep` of Section 1.1.1.3, whereas the relaxed variant calls Algorithm `RelaxedProductStep` of Section 1.1.3.4. In fact, since we work on p-adic integers, the relaxed version uses an implementation of Algorithm $\texttt{Binary\_Mul\_Padic}_p$ from [BHL11, Section 3.2], which is a p-adic integer variant of Algorithm `RelaxedProductStep`.

Furthermore, when the precision is high, we make use of blocks of size 32 or 1024. That means, that at first, we compute the solution f up to precision 32 as $F_0 = f_0 + \cdots + f_{31}\, p^{31}$ with the variant "N". Then, we say that our solution can be seen as a p^{32}-adic integer $F = F_0 + \cdots + F_n p^{32n} + \cdots$ and the algorithm runs with F_0 as the initial condition. Then, each F_n is decomposed in base p to retrieve $f_{32n}, \ldots, f_{32n+31}$. Although it is competitive, the initialization of F can be quite expensive. "BN" means that F is computed with the variant "N", while "BR" means it is with the variant "R". Finally, if the precision is high enough, one may want to compute F with blocks of size 32, and therefore f with blocks of size 1024. "B^{2}N" (resp. "B^{2}R") means that f and F are computed up to precision 32 with the variant "N" and then, the p^{1024}-adic solution is computed with the variant "N" (resp. "R").

The next two tables correspond to timings for computing $B^{-1} \cdot A$ at precision n, with $A, B \in \mathcal{M}_{r\times r}(\mathbb{Z}_p)$. In this case, it is fair to compare our relaxed algorithm with Newton's iteration algorithm because they both have quasi-optimal cost $\tilde{\mathcal{O}}(r^\omega\, n)$. We see that "Relaxed" performs well.

n	4	16	64	2^8	2^{10}	2^{12}	2^{14}	2^{16}
Newton	0.097	0.22	0.89	6.8	59	490	3400	20000
Relaxed	0.15	0.61	3.1	8.1	38	335	1600	14000
Variant	N	N	N	BN	BN	BN	B^{2}N	B^{2}N

Table 3.8. Square matrices of size $r = 8$

n	4	16	64	2^8	2^{10}
Newton	930	2600	14000	140000	1300000
Relaxed	3600	18000	53000	150000	1000000
Variant	N	N	N	BN	BN

Table 3.9. Square matrices of size $r = 128$

Now, we solve integer linear systems and retrieve the solutions over $\mathbb{Q}$, using the rational number reconstruction [GG03, Section 5.10]. We set q as p to the power 2^j and pick at random a square matrix B of size r with coefficients in $M = \{0, \ldots, q-1\}$. We solve $B \cdot C = A$ with a random vector A. Because we deal with q-adic numbers at low precision, we only use the variant "N". We compared with LinBox [Lin08] and IML [CS04] but we do not display the timings of IML within LinBox because they are about 10 times slower. As LinBox and IML are designed for big matrices and small integers, it is not surprising that "Relaxed" performs better on these small matrices with big integers.

j	0	2	4	6	8	10	12
LinBox	1.0	1.4	3.6	25	310	4700	77000
Relaxed	0.10	0.24	0.58	2.1	14	110	760

Table 3.10. Integer linear system of size $r = 4$

j	0	2	4	6	8	10
LinBox	5.9	25	170	1900	27000	480000
Relaxed	24	150	360	2000	14000	90000

Table 3.11. Integer linear system of size $r = 32$

In fact, when $j \leqslant 3$, there is a major overhead coming from the use of Gmp. Indeed, in these cases, we transform q-adic numbers into p-adic numbers, compute up to the necessary precision and call the rational reconstruction.

Acknowledgments

We would like to thank J. van der Hoeven, M. Giusti, G. Lecerf, M. Mezzarobba and É. Schost for their helpful comments and remarks. For their help with LinBox, we thank B. Boyer and J.-G. Dumas.

Chapitre 4

Power series solutions of (q)-differential equations

This chapter is published in the homonym paper with A. Bostan, M. Chowdhurry, B. Salvy and É. Schost in the proceedings of *ISSAC'12* [BCL+12].

We provide algorithms computing power series solutions of a large class of differential or q-differential equations or systems. Their number of arithmetic operations grows linearly with the precision, up to logarithmic terms.

4.1 Introduction

Truncated power series are a fundamental class of objects of computer algebra. Fast algorithms are known for a large number of operations starting from addition, derivative, integral and product and extending to quotient, powering and several more. The main open problem is composition: given two power series f and g, with $g(0) = 0$, known mod x^N, the best known algorithm computing $f(g) \bmod x^N$ has a cost which is roughly that of $\sqrt{N}$ products in precision N; it is not known whether quasi-linear (i.e., linear up to logarithmic factors) complexity is possible in general. Better results are known over finite fields [Ber98, KU11] or when more information on f or g is available. Quasi-linear complexity has been reached when g is a polynomial [BK78], an algebraic series [Hoe02], or belongs to a large class containing for instance the expansions of $\exp(x) - 1$ and $\log(1 + x)$ [BSS08].

One motivation for this work is to deal with the case when f is the solution of a given differential equation. Using the chain rule, a differential equation for $f(g)$ can be derived, with coefficients that are power series. We focus on the case when this equation is linear, since in many cases linearization is possible [BCO+07]. When the order n of the equation is larger than 1, we use the classical technique of converting it into a first-order equation over vectors, so we consider equations of the form

$$x^k \, \delta(F) = A \, F + C, \tag{4.1}$$

where A is an $n \times n$ matrix over the power series ring $\Bbbk[[x]]$ ($\Bbbk$ being the field of coefficients), C and the unknown F are size n vectors over $\Bbbk[[x]]$ and for the moment δ denotes the differential operator $d/d\,x$. The exponent k in (4.1) is a non-negative integer that plays a key role for this equation.

By *solving* such equations, we mean computing a vector F of power series such that (4.1) holds modulo x^N. For this, we need only to compute F polynomial of degree less than $N+1$ (when $k=0$) or N (otherwise). Conversely, when (4.1) has a power series solution, its first N coefficients can be computed by solving (4.1) modulo x^N (when $k\neq 0$) or x^{N-1} (otherwise).

If $k=0$ and the field $\Bbbk$ has characteristic 0, then a formal Cauchy theorem holds and (4.1) has a unique vector of power series solution for any given initial condition. In this situation, algorithms are known that compute the first N coefficients of the solution in quasi-linear complexity [BCO+07]. Also the relaxed algorithm of [Hoe02] applies to this case. In this article, we extend the results of [BCO+07] in three directions:

Singularities We deal with the case when k is positive. A typical example is the computation of the composition $F = f(g)$ when f is Gauss' ${}_2F_1$ hypergeometric series. Although f is a very nice power series

$$f = 1 + \frac{a\,b}{c}\,x + \frac{a\,(a+1)\,b\,(b+1)}{c\,(c+1)}\,\frac{x^2}{2!} + \cdots,$$

we exploit this structure indirectly only. We start from the differential equation

$$x\,(x-1)\,f'' + (x\,(a+b+1) - c)\,f' + a\,b\,f = 0 \tag{4.2}$$

and build up and solve the more complicated

$$\frac{g\,(g-1)}{g'^2}\,F'' + \frac{g'^2\,(g\,(a+b+1)-c) + (g-g^2)\,g''}{g'^3}\,F' + a\,b\,F = 0$$

in the unknown F, g being given, with $g(0)=0$. Equation (4.2) has a leading term that is divisible by x so that Cauchy's theorem does not apply and indeed there does not exist a basis of two power series solutions. This behavior is inherited by the equation for F, so that the techniques of [BCO+07] do not apply — this example is actually already mentioned in [BK78], but the issue with the singularity at 0 was not addressed there. We show in this article how to overcome this singular behavior and obtain a quasi-linear complexity.

Positive characteristic Even when $k=0$, Cauchy's theorem does not hold in positive characteristic and Equation (4.1) may fail to have a power series solution (a simple example is $F'=F$). However, such an equation may have a solution modulo x^N. Efficient algorithms for finding such a solution are useful in conjunction with the Chinese remainder theorem. Other motivations for considering algorithms that work in positive characteristic come from applications in number-theory based cryptology or in combinatorics [BMSS08, BSS08, BS09].

Our objectives in this respect are to overcome the lack of a Cauchy theorem, or of a formal theory of singular equations, by giving conditions that ensure the existence of solutions at the required precisions. More could probably be said regarding the p-adic properties of solutions of such equations (as in [BGVPS05, LS08]), but this is not the purpose of this chapter.

Functional Equations The similarity between algorithms for linear differential equations and for linear difference equations is nowadays familiar to computer algebraists. We thus use the standard technique of introducing $\sigma\colon \Bbbk[[x]] \to \Bbbk[[x]]$ a unitary ring morphism and letting $\delta\colon \Bbbk[[x]] \to \Bbbk[[x]]$ denote a σ-derivation, in the sense that δ is $\Bbbk$-linear and that for all f, g in $\Bbbk[[x]]$, we have

$$\delta(fg) = f\delta(g) + \delta(f)\,\sigma(g).$$

These definitions, and the above equality, carry over to matrices over $\Bbbk[[x]]$. Thus, our goal is to solve the following generalization of (4.1):

$$x^k\,\delta(F) = A\,\sigma(F) + C. \tag{4.3}$$

As above, we are interested in computing a vector F of power series such that (4.3) holds $\bmod x^N$.

Concerning on-line algorithms, the techniques of [Hoe02] already apply to the positive characteristic case. At the beginning of my thesis, the tools to adapt relaxed algorithms for singular equations did not exist. Our method to deal with singular equations was discovered independently at the same time by [Hoe11]. This paper deals with more general recursive power series defined by algebraic, differential equations or a combination thereof. However, this paper does not consider the case of (q)-differential equations and works under more restrictive hypotheses.

One motivation for the generalization to functional equations comes from coding theory. The list-decoding of the folded Reed-Solomon codes [GR08] leads to an equation $Q\,(x, f(x), f\,(q\,x)) = 0$ where Q is a known polynomial. A linearized version of this is of the form (4.3), with $\sigma\colon \phi(x) \mapsto \phi(q\,x)$. In cases of interest we have $k = 1$, and we work over a finite field.

In view of these applications, we restrict ourselves to the following setting:

$$\delta(x) = 1, \qquad \sigma\colon x \mapsto q\,x,$$

for some $q \in \Bbbk \setminus \{0\}$. Then, there are only two possibilities:

- $q = 1$ and $\delta\colon f \mapsto f'$ (*differential* case);
- $q \neq 1$ and $\delta\colon f \mapsto \frac{f(q\,x) - f(x)}{x\,(q-1)}$ (*q-differential* case).

As a consequence, $\delta(1) = 0$ and for all $i \geq 0$, we have

$$\delta(x^i) = \gamma_i\,x^{i-1} \text{ with } \gamma_0 = 0 \text{ and } \gamma_i = 1 + q + \cdots + q^{i-1}\,(i > 0).$$

By linearity, given $f = \sum_{i\geq 0} f_i\,x^i \in \Bbbk[[x]]$,

$$\delta(f) = \sum_{i\geq 1} \gamma_i\,f_i\,x^{i-1}$$

can be computed $\bmod x^N$ in $\mathcal{O}(N)$ operations, as can $\sigma(f)$. Conversely, assuming that $\gamma_1, \ldots, \gamma_n$ are all non-zero in $\Bbbk$, given f of degree at most $n-1$ in $\Bbbk[x]$, there exists a unique g of degree at most n such that $\delta(g) = f$ and $g_0 = 0$; it is given by $g = \sum_{0 \le i \le n-1} f_i/\gamma_{i+1}\, x^{i+1}$ and can be computed in $\mathcal{O}(N)$ operations. We denote it by $g = \int_q f$. In particular, our condition excludes cases where q is a root of unity of low order.

Notation and complexity model We adopt the convention that uppercase letters denote matrices or vectors while lowercase letters denote scalars. The set of $n \times m$ matrices over a ring R is denoted $\mathcal{M}_{n,m}(R)$; when $n = m$, we write $\mathcal{M}_n(R)$. If f is in $\Bbbk[[x]]$, its degree i coefficient is written f_i; this carries over to matrices. The identity matrix is written Id (the size will be obvious from the context). To avoid any confusion, the entry (i, j) of a matrix M is denoted $M^{(i,j)}$.

Our algorithms are sometimes stated with input in $\Bbbk[[x]]$, but it is to be understood that we are given only truncations of A and C and only their first N coefficients will be used.

The costs of our algorithms are measured by the number of arithmetic operations in $\Bbbk$ they use. We let $\mathsf{M}: \mathbb{N} \to \mathbb{N}$ be such that for any ring R, polynomials of degree less than n in $R[x]$ can be multiplied in $\mathsf{M}(n)$ arithmetic operations in R. We assume that $\mathsf{M}(n)$ satisfies the usual assumptions of [GG03, §8.3]; using Fast Fourier Transform, $\mathsf{M}(n)$ can be taken in $\mathcal{O}(n \log(n) \operatorname{loglog}(n))$ [CK91, SS71]. We note $\omega \in (2, 3]$ a constant such that two matrices in $\mathcal{M}_n(R)$ can be multiplied in $\mathcal{O}(n^\omega)$ arithmetic operations in R. The current best bound is $\omega < 2.3727$ ([VW11] following [CW90, Sto10]).

Our algorithms rely on linear algebra techniques; in particular, we have to solve several systems of non-homogeneous linear equations. For U in $\mathcal{M}_n(\Bbbk)$ and V in $\mathcal{M}_{n,1}(\Bbbk)$, we denote by $\mathsf{LinSolve}(U\ X = V)$ a procedure that returns $\perp$ if there is no solution, or a pair F, K, where F is in $\mathcal{M}_{n,1}(\Bbbk)$ and satisfies $U\,F = V$, and $K \in \mathcal{M}_{n,t}(\Bbbk)$, for some $t \le n$, generates the nullspace of U. This can be done in time $\mathcal{O}(n^\omega)$. In the pseudo-code, we adopt the convention that if a subroutine returns $\perp$, the caller returns $\perp$ too (so we do not explicitly handle this as a special case).

Main results Equation (4.3) is linear, non-homogeneous in the coefficients of F, so our output follows the convention mentioned above. We call *generators* of the solution space of Eq. (4.3) at precision N either $\perp$ (if no solution exists) or a pair F, K where $F \in \mathcal{M}_{n,1}(\Bbbk[x])$ and $K \in \mathcal{M}_{n,t}(\Bbbk[x])$ with $t \le n\,N$, such that for $G \in \mathcal{M}_{n,1}(\Bbbk[x])$, with $\deg(G) < N$, $x^k\,\delta(G) = A\,\sigma(G) + C \bmod x^N$ if and only if G can be written $G = F + K\,B$ for some $B \in \mathcal{M}_{t,1}(\Bbbk)$.

Seeing Eq. (4.3) as a linear system, one can obtain such an output using linear algebra in dimension $n\,N$. While this solution always works, we give algorithms of much better complexity, under some assumptions related to the spectrum $\operatorname{Spec} A_0$ of the constant coefficient A_0 of A. First, we simplify our problem: we consider the case $k = 0$ as a special case of the case $k = 1$. Indeed, the equation $\delta(F) = A\,\sigma(F) + C \bmod x^N$ is equivalent to $x\,\delta(F) = P\,\sigma(F) + Q \bmod x^{N+1}$, with $P = xA$ and $Q = xC$. Thus, in our results, we only distinguish the cases $k = 1$ and $k > 1$.

Definition 4.1. *The matrix A_0 has* good spectrum at precision N *when one of the following holds:*

- $k=1$ *and* $\mathrm{Spec}A_0 \cap (q^i\,\mathrm{Spec}A_0 - \gamma_i) = \emptyset$ *for* $1 \le i < N$
- $k>1$, A_0 *is invertible and*
  - $q=1$, $\gamma_1, \ldots, \gamma_{N-k}$ *are non-zero,* $|\mathrm{Spec}A_0| = n$ *and* $\mathrm{Spec}A_0 \subset \Bbbk$;
  - $q \neq 1$ *and* $\mathrm{Spec}A_0 \cap q^i\,\mathrm{Spec}A_0 = \emptyset$ *for* $1 \le i < N$.

In the classical case when $\Bbbk$ has characteristic 0 and $q=1$, if $k=1$, A_0 has good spectrum when no two eigenvalues of A_0 differ by a non-zero integer (this is e.g. the case when $A_0=0$, which is essentially the situation of Cauchy's theorem; this is also the case in our ${}_2F_1$ example whenever $c\mathrm{val}(g)$ is not an integer, since $\mathrm{Spec}A_0 = \{0, \mathrm{val}(g)\,(1-c)-1\}$).

These conditions could be slightly relaxed, using gauge transformations (see [Bal00, Ch. 2] and [BBP10, BP99]). Also, for $k > 1$ and $q = 1$, we could drop the assumption that the eigenvalues are in $\Bbbk$, by replacing $\Bbbk$ by a suitable finite extension, but then our complexity estimates would only hold in terms of number of operations in this extension.

As in the non-singular case [BCO+07], we develop two approaches. The first one is a divide-and-conquer method. The problem is first solved at precision $N/2$ and then the computation at precision N is completed by solving another problem of the same type at precision $N/2$. This leads us to the following result, proved in Section 4.2 (see also that section for comparison to previous work). In all our cost estimates, we consider k constant, so it is absorbed in the big-Os.

Theorem 4.2. *Algorithm 4.2 computes generators of the solution space of Eq. (4.3) at precision N by a divide-and-conquer approach. Assuming A_0 has good spectrum at precision N, it performs in time $\mathcal{O}(n^\omega\,\mathsf{M}(N)\log(N))$. When either $k>1$ or $k=1$ and $q^i\,A_0 - \gamma_i\,\mathsf{Id}$ is invertible for $0 \le i < N$, this drops to $\mathcal{O}(n^2\,\mathsf{M}(N)\log(N) + n^\omega\,N)$.*

Our second algorithm behaves better with respect to N, with cost in $\mathcal{O}(\mathsf{M}(N))$ only, but it always involves polynomial matrix multiplications. Since in many cases the divide-and-conquer approach avoids these multiplications, the second algorithm becomes preferable for rather large precisions.

In the differential case, when $k=0$ and the characteristic is 0, the algorithms in [BCO+07, BK78] compute an invertible matrix of power series solution of the homogeneous equation by a Newton iteration and then recover the solution using variation of the constant. In the more general context we are considering here, such a matrix does not exist. However, it turns out that an associated equation that can be derived from (4.3) admits such a solution. Section 4.3 describes a variant of Newton's iteration to solve it and obtains the following.

Theorem 4.3. *Assuming A_0 has good spectrum at precision N, one can compute generators of the solution space of Eq. (4.3) at precision N by a Newton-like iteration in time $\mathcal{O}(n^\omega\,\mathsf{M}(N) + n^\omega\log(n)\,N)$.*

To the best of our knowledge, this is the first time such a low complexity is reached for this problem. Without the good spectrum assumption, however, we cannot guarantee that this algorithm succeeds, let alone control its complexity.

4.2 Divide-and-Conquer

The classical approach to solving (4.3) is to proceed term-by-term by coefficient extraction. Indeed, we can rewrite the coefficient of degree i in this equation as

$$R_i F_i = \Delta_i, \tag{4.4}$$

where Δ_i is a vector that can be computed from A, C and all previous F_j (and whose actual expression depends on k), and R_i is as follows:

$$\begin{cases} R_i = (q^i A_0 - \gamma_i \mathsf{Id}) & \text{if } k = 1 \\ R_i = q^i A_0 & \text{if } k > 1. \end{cases}$$

Ideally, we wish that each such system determines F_i uniquely that is, that R_i be a unit. For $k = 1$, this is the case when i is not a root of the *indicial equation* $\det(q^i A_0 - \gamma_i \mathsf{Id}) = 0$. For $k > 1$, either this is the case for all i (when A_0 is invertible) or for no i. In any case, we let $\mathcal{R}$ be the set of indices $i \in \{0, \ldots, N-1\}$ such that $\det(R_i) = 0$; we write $\mathcal{R} = \{j_1 < \ldots < j_r\}$, so that $r = |\mathcal{R}|$.

Even when $\mathcal{R}$ is empty, so the solution is unique, this approach takes quadratic time in N, as computing each individual Δ_i takes linear time in i. To achieve quasi-linear time, we split the resolution of Eq. (4.3) mod x^N into two half-sized instances of the problem; at the leaves of the recursion tree, we end up having to solve the same Eq. (4.4).

When $\mathcal{R}$ is empty, the algorithm is simple to state (and the cost analysis simplifies; see the comments at the end of this section). Otherwise, technicalities arise. We treat the cases $i \in \mathcal{R}$ separately, by adding placeholder parameters for all corresponding coefficients of F (this idea is already in [BBP10, BP99]; the algorithms in these references use a finer classification when $k > 1$, by means of a suitable extension of the notion of indicial polynomial, but take quadratic time in N).

Let $\mathbf{f}_{1,1}$, ..., $\mathbf{f}_{n,r}$ be $n\,r$ new indeterminates over $\Bbbk$ (below, all boldface letters denote expressions involving these formal parameters). For $\rho = 1, \ldots, r$, we define the vector $\mathbf{F}_{j_\rho}$ with entries $\mathbf{f}_{1,\rho}, \ldots, \mathbf{f}_{n,\rho}$ and we denote by $\mathscr{L}$ the set of all vectors

$$\mathbf{F} = \varphi_0 + \varphi_1 \mathbf{F}_{j_1} + \cdots + \varphi_r \mathbf{F}_{j_r},$$

with φ_0 in $\mathcal{M}_{n,1}(\Bbbk[x])$ and each φ_ℓ in $\mathcal{M}_n(\Bbbk[x])$ for $1 \le \ell \le r$. We also define $\mathscr{L}_i$ the subspace of vectors of the form

$$\mathbf{F} = \varphi_0 + \varphi_1 \mathbf{F}_{j_1} + \cdots + \varphi_{\mu(i)} \mathbf{F}_{\mu(i)},$$

where $\mu(i)$ is defined as the index of the largest element $j_\ell \in \mathcal{R}$ such that $j_\ell < i$; if no such element exist (for instance when $i = 0$), we let $\mu(i) = 0$. A *specialization* S: $\mathscr{L} \to \mathcal{M}_{n,1}(\Bbbk[x])$ is simply an evaluation map defined by $f_{i,\ell} \mapsto f_{i,\ell}$ for all i, ℓ, for some choice of $(f_{i,\ell})$ in $\Bbbk^{nr}$.

We extend δ and σ to such vectors, by letting $\delta(\mathbf{f}_{i,\ell}) = 0$ and $\sigma(\mathbf{f}_{i,\ell}) = \mathbf{f}_{i,\ell}$ for all i, ℓ, so that we have, for $\mathbf{F}$ in $\mathscr{L}$

$$\delta(\mathbf{F}) = \delta(\varphi_0) + \delta(\varphi_1)\,\mathbf{F}_{j_1} + \cdots + \delta(\varphi_r)\,\mathbf{F}_{j_r},$$

and similarly for $\sigma(\mathbf{F})$.

Algorithm 4.1
Recursive Divide-and-conquer $\mathsf{RDAC}(A, \mathbf{C}, i, N, k)$

Input: $A \in \mathcal{M}_n(\Bbbk[[x]]), \mathbf{C} \in \mathscr{L}_i, i \in \mathbb{N}, N \in \mathbb{N} \setminus \{0\}, k \in \mathbb{N} \setminus \{0\}$
Output: $\mathbf{F} \in \mathscr{L}_{i+N}$

if $N = 1$
  if $(k = 1)$ **then** $R_i := q^i A_0 - \gamma_i \mathsf{Id}$ **else** $R_i := q^i A_0$
  if $(\det(R_i) = 0)$ **then return** $\mathbf{F}_i$ **else return** $-R_i^{-1}\mathbf{C}_0$
else
  $m := \lceil N/2 \rceil$
  $\mathbf{H} := \mathsf{RDAC}(A, \mathbf{C}, i, m, k)$
  $\mathbf{D} := (\mathbf{C} - x^k\,\delta(\mathbf{H}) + (q^i A - \gamma_i x^{k-1}\mathsf{Id})\,\sigma(\mathbf{H}))$ div x^m
  $\mathbf{K} := \mathsf{RDAC}(A, \mathbf{D}, i + m, N - m, k)$
  return $\mathbf{H} + x^m \mathbf{K}$

The main divide-and-conquer algorithm first computes $\mathbf{F}$ in $\mathscr{L}$, by simply skipping all equations corresponding to indices $i \in \mathcal{R}$; it is presented in Algorithm 4.2. In a second step, we resolve the indeterminacies by plain linear algebra. For $i \geq 0$, and $\mathbf{F}, \mathbf{C}$ in $\mathscr{L}$, we write

$$E(\mathbf{F}, \mathbf{C}, i) = x^k\,\delta(\mathbf{F}) - ((q^i A - \gamma_i x^{k-1}\mathsf{Id})\,\sigma(\mathbf{F}) + \mathbf{C}).$$

In particular, $E(\mathbf{F}, \mathbf{C}, 0)$ is a parameterized form of Eq. (4.3). The key to the divide-and-conquer approach is to write $\mathbf{H} = \mathbf{F} \bmod x^m$, $\mathbf{K} = \mathbf{F}$ div x^m and $\mathbf{D} = (\mathbf{C} - E(\mathbf{H}, \mathbf{C}, i))$ div x^m. Using the equalities

$$x^k\,\delta(\mathbf{F}) = x^k\,\delta(\mathbf{H}) + x^{m+k}\,\delta(\mathbf{K}) + \gamma_m\,x^{m+k-1}\,\sigma(\mathbf{K})$$

and $\gamma_{i+m} = \gamma_m + q^m\,\gamma_i$, a quick computation shows that

$$E(\mathbf{F}, \mathbf{C}, i) = (E(\mathbf{H}, \mathbf{C}, i) \bmod x^m) + x^m\,E(\mathbf{K}, \mathbf{D}, i + m). \tag{4.5}$$

Lemma 4.4. *Let A be in $\mathcal{M}_n(\Bbbk[x])$ and $\mathbf{C}$ in $\mathscr{L}_i$, and let $\mathbf{F} = \mathsf{RDAC}(A, \mathbf{C}, i, M, k)$ with $i + M \leq N$. Then:*

1. *$\mathbf{F}$ is in $\mathscr{L}_{i+M}$;*

2. *for $j \in \{0,\dots,M-1\}$ such that $i+j \notin \mathcal{R}$, the equality* $\mathrm{coeff}(E(\mathbf{F},\mathbf{C},i),x^j)=0$ *holds;*

3. *if C and F in $\mathcal{M}_{n,1}(\Bbbk[x])$ with $\deg F < M$ are such that $E(F,C,i)=0 \bmod x^M$ and there exists a specialization $S\colon \mathscr{L}_i \to \mathcal{M}_{n,1}(\Bbbk[x])$ such that $C = S(\mathbf{C})$, there exists a specialization $S'\colon \mathscr{L}_{i+M} \to \mathcal{M}_{n,1}(\Bbbk[x])$ which extends S and such that $F = S(\mathbf{F})$.*

$\mathbf{F}$ *is computed in time $\mathcal{O}((n^2 + r\,n^\omega)\,\mathsf{M}(M)\log(M) + n^\omega\,M)$.*

Proof. The proof is by induction on M.

Proof of 1. For $M=1$, we distinguish two cases. If $i \in \mathcal{R}$, say $i = j_\ell$, we return $\mathbf{F}_i = \mathbf{F}_{j_\ell}$. In this case, $\mu(i+1) = \ell$, so our claim holds. If $i \notin \mathcal{R}$, because $\mathbf{C}_0 \in \mathscr{L}_i$, the output is in $\mathscr{L}_i$ as well. This proves the case $M=1$.

For $M > 1$, we assume the claim to hold for all $(i,\ M')$, with $M' < M$. By induction, $\mathbf{H} \in \mathscr{L}_{i+m}$ and $\mathbf{K} \in \mathscr{L}_{i+M}$. Thus, $\mathbf{D} \in \mathscr{L}_{i+m}$ and the conclusion follows.

Proof of 2. For $M=1$, if $i \in \mathcal{R}$, the claim is trivially satisfied. Otherwise, we have to verify that the constant term of $E(\mathbf{F},\mathbf{C},i)$ is zero. In this case, the output $\mathbf{F}$ is reduced to its constant term $\mathbf{F}_0$, and the constant term of $E(\mathbf{F},\mathbf{C},i)$ is (up to sign) $R_i\mathbf{F}_0 + \mathbf{C}_0 = 0$, so we are done.

For $M>1$, we assume that the claim holds for all (i,M'), with $M'<M$. Take j in $\{0,\dots,M-1\}$. If $j<m$, we have $\mathrm{coeff}(E(\mathbf{F},\mathbf{C},i),x^j) = \mathrm{coeff}(E(\mathbf{H},\mathbf{C},i),x^j)$; since $i+j \notin \mathcal{R}$, this coefficient is zero by assumption. If $m \le j$, we have $\mathrm{coeff}(E(\mathbf{F},\mathbf{C},i),$ $x^j) = \mathrm{coeff}(E(\mathbf{K},\mathbf{D},i),x^{j-m})$. Now, $j+i \notin \mathcal{R}$ implies that $(j-m)+(i+m) \notin \mathcal{R}$, and $j-m < M-m$, so by induction this coefficient is zero as well.

Proof of 3. For $M=1$, if $i \in \mathcal{R}$, say $i = j_\ell$, we have $\mathbf{F} = \mathbf{F}_{j_\ell}$, whereas F has entries in $\Bbbk$; this allows us to define S'. When $i \notin \mathcal{R}$, we have $F = S(\mathbf{F})$, so the claim holds as well. Thus, we are done for $M=1$.

For $M>1$, we assume our claim for all (i,M') with $M'<M$. Write $H = F \bmod x^m$, $K = F \operatorname{div} x^m$ and $D = (C - x^k\,\delta(H) + (q^i\,A - \gamma_i\,x^{k-1}\,\mathsf{Id})\,\sigma(H)) \operatorname{div} x^m$. Then, (4.5) implies that $E(H,C,i) = 0 \bmod x^m$ and $E(K,D,i+m) = 0 \bmod x^{M-m}$. The induction assumption shows that H is a specialization of $\mathbf{H}$, say $H = S'(\mathbf{H})$ for some S': $\mathscr{L}_{i+m} \to \mathcal{M}_{n,1}(\Bbbk[x])$ which extends S. In particular, $D = S'(\mathbf{D})$. The induction assumption also implies that there exist an extension $S''\colon \mathscr{L}_{i+m} \to \mathcal{M}_{n,1}(\Bbbk[x])$ of S', and thus of S, such that $K = S''(\mathbf{K})$. Then $F = S''(\mathbf{F})$, so we are done.

For the complexity analysis, the most expensive part of the algorithm is the computation of $\mathbf{D}$. At the inner recursion steps, the bottleneck is the computation of $A\ \sigma(\mathbf{H})$, where $\mathbf{H}$ has degree less than M and A can be truncated mod x^M (the higher degree terms have no influence in the subsequent recursive calls). Computing $\sigma(\mathbf{H})$ takes time $\mathcal{O}(N\ (n\ +\ r\ n^2))$ and the product is done in time $\mathcal{O}((n^2 + r\,n^\omega)\,\mathsf{M}(M))$; recursion leads to a factor $\log(M)$. The base cases use $\mathcal{O}(M)$ matrix inversions of cost $\mathcal{O}(n^\omega)$ and $\mathcal{O}(M)$ multiplications, each of which takes time $\mathcal{O}(r\,n^\omega)$. □

The second step of the algorithm is plain linear algebra: we know that the output of the previous algorithm satisfies our main equation for all indices $i \notin \mathcal{R}$, so we conclude by forcing the remaining ones to zero.

Algorithm 4.2
Divide-and-Conquer $\mathsf{DAC}(A, C, N, k)$

Input: $A \in \mathcal{M}_n(\Bbbk[[x]]), C \in \mathcal{M}_{n,1}(\Bbbk[[x]]),\ N \in \mathbb{N} \setminus \{0\}, k \in \mathbb{N} \setminus \{0\}$
Output: Generators of the solution space of $x^k\,\delta(F) = A\,\sigma(F) + C$ at precision N.

$\mathbf{F} := \mathsf{RDAC}(A, C, 0, N, k)$
($\mathbf{F}$ has the form $\varphi_0 + \varphi_1\,\mathbf{F}_{j_1} + \cdots + \varphi_r\,\mathbf{F}_{j_r}$)
$\mathbf{T} := x^k\,\delta(\mathbf{F}) - A\,\sigma(\mathbf{F}) - C \bmod x^N$
$\Gamma := (\mathbf{T}_i^{(j)}, i \in \mathcal{R}, j = 1, \ldots, n)$
$\Phi, \Delta := \mathsf{LinSolve}(\Gamma = 0)$
$M := [\varphi_1, \ldots, \varphi_r]$
return $\varphi_0 + M\,\Phi, M\,\Delta$

Proposition 4.5. *On input A, C, N, k as specified in Algorithm 4.2, this algorithm returns generators of the solution space of (4.3) mod x^N in time $\mathcal{O}((n^2 + r\,n^\omega)\,\mathsf{M}(N)\log(N) + r^2\,n^\omega\,N + r^\omega\,n^\omega)$.*

Proof. The first claim is a direct consequence of the construction above, combined with Lemma 4.4. For the cost estimate, we need to take into account the computation of $\mathbf{T}$, the linear system solving, and the final matrix products. The computation of $\mathbf{T}$ fits in the same cost as that of $\mathbf{D}$ in Algorithm 4.1, so no new contribution comes from here. Solving the system $\Gamma = 0$ takes time $\mathcal{O}((r\,n)^\omega)$. Finally, the product $[\varphi_1 \cdots \varphi_r]\,\Delta$ involves an $n \times (r\,n)$ matrix with entries of degree N and an $(r\,n) \times t$ constant matrix, with $t \le r\,n$; proceeding coefficient by coefficient, and using block matrix multiplication in size n, the cost is $\mathcal{O}(r^2\,n^\omega\,N)$. □

When all matrices R_i are invertible, the situation becomes considerably simpler: $r = 0$, the solution space has dimension 0, there is no need to introduce formal parameters, the cost drops to $\mathcal{O}(n^2\ \mathsf{M}(N)\ \log\ (N) + n^\omega\ N)$ for Lemma 4.4, and Proposition 4.5 becomes irrelevant.

When A_0 has good spectrum at precision N, we may not be able to ensure that $r = 0$, but we always have $r \le 1$. Indeed, when $k = 1$, the good spectrum condition implies that for all $0 \le i < N$ and for $j \in \mathbb{N}$, the matrices R_i and R_j have disjoint spectra so that at most one of them can be singular. For $k > 1$, the good spectrum condition implies that all R_i are invertible, whence $r = 0$. This proves Thm. 4.2.

Previous work. As said above, Barkatou and Pflügel [BP99], then Barkatou, Broughton and Pflügel [BBP10], already gave algorithms that solve such equations term-by-term, introducing formal parameters to deal with cases where the matrix R_i is singular. These algorithms handle some situations more finely than we do (e.g., the cases $k \ge 2$), but take quadratic time; our algorithm can be seen as a divide-and-conquer version of these results.

In the particular case $q \neq 1$, $n=1$ and $r=0$, another forerunner to our approach is Brent and Traub's divide-and-conquer algorithm [BT80]. That algorithm is analyzed for a more general σ, of the form $\sigma(x) = x\,q(x)$, as such, they are more costly than ours; when q is constant, we essentially end up with the approach presented here.

Let us finally mention van der Hoeven's paradigm of relaxed algorithms [Hoe02, Hoe09, Hoe11], which allows one to solve systems such as (4.3) in a term-by-term fashion, but in quasi-linear time. The cornerstone of this approach is fast *relaxed multiplication*, otherwise known as *online multiplication*, of power series.

In [Hoe02, Hoe03], van der Hoeven offers two relaxed multiplication algorithms (the first one being similar to that of [FS74]); both take time $\mathcal{O}(\mathsf{M}(n)\,\log\,(n))$. When $r=0$, this yields a complexity similar to Prop. 4.5 to solve Eq. (4.3), but it is unknown to us how this carries over to arbitrary r.

When $r = 0$, both our divide-and-conquer approach and the relaxed one can be seen as "fast" versions of quadratic time term-by-term extraction algorithms. It should appear as no surprise that they are related: as it turns out, at least in simple cases (with $k=1$ and $n=1$), using the relaxed multiplication algorithm of [Hoe03] to solve Eq. (4.3) leads to doing exactly the same operations as our divide-and-conquer method, without any recursive call. We leave the detailed analysis of these observations to future work.

For suitable "nice" base fields (e.g., for fields that support Fast Fourier Transform), the relaxed multiplication algorithm in [Hoe02] was improved in [Hoe07, Hoe12], by means of a reduction of the $\log\,(n)$ overhead. This raises the question whether such an improvement is available for divide-and conquer techniques.

4.3 Newton Iteration

4.3.1 Gauge Transformation

Let F be a solution of Eq. (4.3). To any invertible matrix $W \in \mathcal{M}_n(\Bbbk[x])$, we can associate the matrix $Y = W^{-1}\,F \in \mathcal{M}_n(\Bbbk[[x]])$. We are going to choose W in such a way that Y satisfies an equation simpler than (4.3). The heart of our contribution is the efficient computation of such a W.

Lemma 4.6. *Let $W \in \mathcal{M}_n(\Bbbk[x])$ be invertible in $\mathcal{M}_n(\Bbbk[[x]])$ and let $B \in \mathcal{M}_n(\Bbbk[x])$ be such that*

$$B = W^{-1}\,(x^k\,\delta(W) - A\,\sigma(W)) \bmod x^N. \tag{4.6}$$

Then F in $\mathcal{M}_{n,1}(\Bbbk[x])$ satisfies

$$x^k\,\delta(F) = A\,\sigma(F) + C \bmod x^N \tag{4.7}$$

if and only if $Y = W^{-1} F$ *satisfies*

$$x^k \delta(Y) = B\,\sigma(Y) + W^{-1} C \bmod x^N. \tag{4.8}$$

Proof. Differentiating the equality $F = WY$ gives

$$x^k \delta(F) = x^k \delta(W)\,\sigma(Y) + x^k W \delta(Y).$$

Since $x^k \delta(W) = A\,\sigma(W) - WB \bmod x^N$, we deduce

$$x^k \delta(F) - A\,\sigma(F) - C = W\,(x^k \delta(Y) - B\,\sigma(Y) - W^{-1} C) \bmod x^N.$$

Since W is invertible, the conclusion follows. □

The systems (4.3) and (4.8) are called equivalent under the gauge transformation $Y = WF$. Solving (4.3) is thus reduced to finding a simple B such that (4.8) can be solved efficiently and such that the equation

$$x^k \delta(W) = A\,\sigma(W) - WB \bmod x^N \tag{4.9}$$

that we call *associated* to (4.3) has an *invertible* matrix W solution that can be computed efficiently too.

As a simple example, consider the differential case, with $k = 1$. Under the good spectrum assumption, it is customary to choose $B = A_0$, the constant coefficient of A. In this case, the matrix W of the gauge transformation must satisfy

$$x\,W' = A\,W - W\,A_0 \bmod x^N.$$

It is straightforward to compute the coefficients of W one after the other, as they satisfy $W_0 = \mathsf{Id}$ and, for $i > 0$,

$$(A_0 - i\,\mathsf{Id})\,W_i - W_i\,A_0 = -\sum_{j<i} A_{i-j} W_j.$$

However, using this formula leads to a quadratic running time in N. The Newton iteration presented in this section computes W in quasi-linear time.

4.3.2 Polynomial Coefficients

Our approach consists in reducing efficiently the resolution of (4.3) to that of an equivalent equation where the matrix A of power series is replaced by a matrix B of polynomials of low degree. This is interesting because the latter can be solved in *linear* complexity by extracting coefficients. This subsection describes the resolution of the equation

$$x^k \delta(Y) = P\sigma(Y) + Q, \tag{4.10}$$

where P is a polynomial matrix of degree less than k.

Algorithm 4.3 PolCoeffsDE(P, Q, k, N)

Input: $P \in \mathcal{M}_n(\Bbbk[x])$ of degree less than k,
$Q \in \mathcal{M}_{n,1}(\Bbbk[[x]])$, $N \in \mathbb{N} \setminus \{0\}$, $k \in \mathbb{N} \setminus \{0\}$
Output: Generators of the solution space of
$x^k\,\delta(Y) = P\sigma(Y) + Q$ at precision N.

for $i = 0, \ldots, N-1$
  $C := Q_i + (P_1\,q^{i-1}\,Y_{i-1} + \ldots + P_{k-1}\,q^{i-k+1}\,Y_{i-k+1})$
  if $(k = 1)$
    $Y_i, M_i := \mathsf{LinSolve}((\gamma_i\,\mathsf{Id} - q^i\,P_0)\,X = C)$
  else
    $Y_i, M_i := \mathsf{LinSolve}(-q^i\,P_0\,X = C - \gamma_{i-k+1}\,Y_{i-k+1})$
return $Y_0 + \ldots + Y_{N-1}\,x^{N-1}$, $[M_0\ M_1\,x \cdots M_{N-1}\,x^{N-1}]$

Lemma 4.7. *Suppose that P_0 has good spectrum at precision N. Then Algorithm 4.3 computes generators of the solution space of Eq. (4.10) at precision N in time $\mathcal{O}(n^\omega\,N)$, with $M \in \mathcal{M}_{n,t}(\Bbbk)$ for some $t \le n$.*

Proof. Extracting the coefficient of x^i in Eq. (4.10) gives

$$\gamma_{i-k+1}\,Y_{i-k+1} = q^i\,P_0\,Y_i + \ldots + q^{i-k+1}\,P_{k-1}\,Y_{i-k+1} + Q_i.$$

In any case, the equation to be solved is as indicated in the algorithm. For $k = 1$, we actually have $C = Q_i$ for all i, so all these systems are independent. For $k > 1$, the good spectrum condition ensures that the linear system has full rank for all values of i, so all M_i are empty. For each i, computing C and solving for Y_i is performed in $\mathcal{O}(n^\omega)$ operations, whence the announced complexity. □

4.3.3 Computing the Associated Equation

Given $A \in \mathcal{M}_n(\Bbbk[[x]])$, we are looking for a matrix B with polynomial entries of degree less than k such that the associated Equation (4.9), which does not depend on the non-homogeneous term C, has an invertible matrix solution.

In this article, we content ourselves with a simple version of the associated equation where we choose B in such a way that (4.9) has an invertible solution $V \bmod x^k$; thus, V and B must satisfy $A\ \sigma(V) = V\,B \bmod x^k$. The invertible matrix V is then lifted at higher precision by Newton iteration (Algorithm 4.6) under regularity conditions that depend on the spectrum of A_0. Other cases can be reduced to this setting by the polynomial gauge transformations that are used in the computation of formal solutions [BBP10, Was65].

When $k = 1$ or $q \neq 1$, the choice

$$B = A \bmod x^k, \quad V = \mathsf{Id}$$

solves our constraints and is sufficient to solve the associated equation. When $q=1$, $k>1$ (in particular when the point 0 is an irregular singular point of the equation), this is not be the case anymore. In that case, we use a known technique called the *splitting lemma* to prepare our equation. See for instance [Bal00, Ch. 3.2] and [BBP10] for details and generalizations.

Lemma 4.8. (Splitting Lemma) *Suppose that $k>1$, that $|\mathrm{Spec}A_0|=n$ and that $\mathrm{Spec}A_0 \subset \Bbbk$. Then one can compute in time $\mathcal{O}(n^\omega)$ matrices V and B of degree less than k in $\mathcal{M}_n(\Bbbk[x])$ such that the following holds: V_0 is invertible; B is diagonal; $AV = VB \bmod x^k$.*

Proof. We can assume that A_0 is diagonal: if not, we let P be in $\mathcal{M}_n(\Bbbk)$ such that $D = P^{-1}AP$ has a diagonal constant term; we find V using D instead of A, and replace V by PV. Computing P and PV takes time $\mathcal{O}(n^\omega)$, since as per convention, k is considered constant in the cost analyses.

Then, we take $B_0 = A_0$ and $V_0 = \mathsf{Id}$. For $i>0$, we have to solve $A_0 V_i - V_i A_0 - B_i = \Delta_i$, where Δ_i can be computed from $A_1, ..., A_i$ and $B_1, ..., B_{i-1}$ in time $\mathcal{O}(n^\omega)$. We set the diagonal of V_i to 0. Since A_0 is diagonal, the diagonal B_i is then equal to the diagonal of Δ_i, up to sign. Then the entry (ℓ, m) in our equation reads $(r_\ell - r_m)\, V_i^{(\ell,m)} = \Delta_i^{(\ell,m)}$, with $r_1, ..., r_n$ the (distinct) eigenvalues of A_0. This can always be solved, in a unique way. The total time is $\mathcal{O}(n^\omega)$. □

4.3.4 Solving the Associated Equation

Once B and V are determined as in §4.3.3, we compute a matrix W that satisfies the associated Equation (4.9); this eventually allows us to reduce (4.3) to an equation with polynomial coefficients. This computation of W is performed efficiently using a suitable version of Newton iteration for Eq. (4.9); it computes a sequence of matrices whose precision is roughly doubled at each stage. This is described in Algorithm 4.6; our main result in this section is the following.

Proposition 4.9. *Suppose that A_0 has good spectrum at precision N. Then, given a solution of the associated equation mod x^k, invertible in $\mathcal{M}_n(\Bbbk[[x]])$, Algorithm 4.6 computes a solution of that equation $\bmod x^N$, also invertible in $\mathcal{M}_n(\Bbbk[[x]])$, in time $\mathcal{O}(n^\omega\, \mathsf{M}(N) + n^\omega \log(n)\, N)$.*

Before proving this result, we show how to solve yet another type of equations that appear in an intermediate step:

$$x^k\, \delta(U) = B\, \sigma(U) - U\, B + \Gamma \bmod x^N, \tag{4.11}$$

where all matrices involved have size $n \times n$, with $\Gamma = 0 \bmod x^m$. This is dealt with by Algorithm 4.4 when $k=1$ or $q \neq 1$ and Algorithm 4.5 otherwise.

For Algorithm 4.4, remember that $B = A \bmod x^k$. The algorithm uses a routine $\mathsf{Sylvester}$ solving *Sylvester equations*. Given matrices Y, V, Z in $\mathcal{M}_n(\Bbbk)$, we are looking for X in $\mathcal{M}_n(\Bbbk)$ such that $YX - XV = Z$. When (Y, V) have disjoint spectra, this system admits a unique solution, which can be computed $\mathcal{O}(n^\omega \log(n))$ operations in $\Bbbk$ [Kir01].

Algorithm 4.4
Solving Eq. (4.11) when $k=1$ or $q\neq 1$ $\quad$ DiffSylvester(Γ, m, N)

Input: $\Gamma \in x^m \mathcal{M}_n(\Bbbk[[x]]), m \in \mathbb{N}\setminus\{0\}, N \in \mathbb{N}\setminus\{0\}$
Output: $U \in x^{m-k}\mathcal{M}_n(\Bbbk[x])$ solution of (4.11).

for $i = m, ..., N-1$
  $C := (B_1\, q^{i-1} U_{i-1} + ... + B_{k-1}\, q^{i-k+1} U_{i-k+1})$
    $-(U_{i-1} B_1 + ... + U_{i-k+1} B_{k-1}) + \Gamma_i$
  if $(k=1)$
    $U_i :=$ Sylvester$(X B_0 + (\gamma_i \,\mathsf{Id} - q^i B_0) X = C)$
  else
    $U_i :=$ Sylvester$(X B_0 - q^i B_0 X = C - \gamma_{i-k+1} U_{i-k+1})$
return $U_m x^m + ... + U_{N-1} x^{N-1}$

Lemma 4.10. *Suppose that $k=1$ or $q\neq 1$ and that A_0 has good spectrum at precision N. If $\Gamma = 0 \bmod x^m$, with $k \leq m < N$, then Algorithm 4.4 computes a solution U to Eq. (4.11) that satisfies $U = 0 \bmod x^{m-k+1}$ in time $\mathcal{O}(n^\omega \log(n)\, N)$.*

Proof. Extracting the coefficient of x^i in (4.11) gives

$$\gamma_{i-k+1} U_{i-k+1} = q^i B_0 U_i - U_i B_0 + C,$$

with C as defined in Algorithm 4.4. In both cases $k=1$ and $k>1$, this gives a Sylvester equation for each U_i, of the form given in the algorithm. Since $B_0 = A_0$, the spectrum assumption on A_0 implies that these equations all have a unique solution. Since Γ is $0 \bmod x^m$, so is U (so we can start the loop at index m). The total running time is $\mathcal{O}(n^\omega \log(n)\, N)$ operations in $\Bbbk$. □

Algorithm 4.5
Solving Eq. (4.11) when $k>1$ or $q=1$ $\quad$ DiffSylvesterDifferential(Γ, m, N)

Input: $\Gamma \in x^m \mathcal{M}_n(\Bbbk[[x]]), m \in \mathbb{N}\setminus\{0\}, N \in \mathbb{N}\setminus\{0\}$
Output: $U \in x^{m-k}\mathcal{M}_n(\Bbbk[x])$ solution of (4.11).

for $i = 1, ..., n$
  for $j = 1, ..., n$
    if $(i=j)$
      $U^{(i,i)} := x^k \int (x^{-k}\Gamma^{(i,i)}) \bmod x^N$
    else
      $U^{(i,j)} :=$ PolCoeffsDE$(B^{(i,i)} - B^{(j,j)}, \Gamma^{(i,j)}, k, N)$
return U

This approach fails in the differential case ($q=1$) when $k>1$, since then the Sylvester systems are all singular. Algorithm 4.5 deals with this issue, using the fact that in this case, B is diagonal, and satisfies the conditions of Lemma 4.8.

Lemma 4.11. *Suppose that $k>1$, $q=1$ and that A_0 has good spectrum at precision N. If $\Gamma=0 \bmod x^m$, with $k\leq m<N$, then Algorithm 4.5 computes a solution U to Eq. (4.11) that satisfies $U=0 \bmod x^{m-k+1}$ in time $\mathcal{O}(n^2 N)$.*

Proof. Since B is diagonal, the (i,j)th entry of (4.11) is

$$x^k \delta(U^{(i,j)}) = (B^{(i,i)} - B^{(j,j)})\, U^{(i,j)} + \Gamma^{(i,j)} \bmod x^N.$$

When $i=j$, $B^{(i,i)}-B^{(j,j)}$ vanishes. After dividing by x^k, we simply have to compute an integral, which is feasible under the good spectrum assumption (we have to divide by the non-zero $\gamma_1=1, \ldots, \gamma_{N-k}=N-k$). When $i\neq j$, the conditions ensure that Lemma 4.7 applies (and since $k>1$, the solution is unique, as pointed out in its proof). □

We now prove the correctness of Algorithm 4.6 for Newton iteration. Instead of doubling the precision at each step, there is a slight loss of $k-1$.

Algorithm 4.6
Newton iteration for Eq. (4.9) NewtonAE(V, N)

Input: $V \in \mathcal{M}_n(\Bbbk[x])$ solution of (4.9) $\bmod x^k$ invertible in $\mathcal{M}_n(\Bbbk[[x]])$, $N \in \mathbb{N} \setminus \{0\}$
Output: $W \in \mathcal{M}_n(\Bbbk[x])$ solution of (4.9) $\bmod x^N$ invertible in $\mathcal{M}_n(\Bbbk[[x]])$, with $W = V \bmod x^k$

```
if (N ⩽ k)
    return V
else
    m := ⌈(N + k − 1)/2⌉
    H := NewtonAE(V, m)
    R := x^k δ(H) − A σ(H) + H B
    if (k = 1) or (q ≠ 1)
        U := DiffSylvester(−H^{−1} R, m, N)
    else
        U := DiffSylvesterDifferential(−H^{−1} R, m, N)
    return H + H U
```

Lemma 4.12. *Let $m \geq k$ and let $H \in \mathcal{M}_n(\Bbbk[x])$ be invertible in $\mathcal{M}_n(\Bbbk[[x]])$ and satisfy (4.9) $\bmod x^m$. Let N be such that $m \leq N \leq 2m-k+1$. Let R and U be as in Algorithm 4.6 and suppose that A_0 has good spectrum at precision N.*

Then $H + HU$ is invertible in $\mathcal{M}_n(\Bbbk[[x]])$ and satisfies the associated equation $\bmod x^N$. Given H, U can be computed in time $\mathcal{O}(n^\omega \mathsf{M}(N) + n^\omega \log(n) N)$.

Proof. By hypothesis, $R=0 \bmod x^m$. Then

$$\begin{aligned} x^k \delta(H+HU) - A\,\sigma(H+HU) + (H+HU)\,B \\ &= (x^k \delta(H) - A\,\sigma(H) + H\,B)\,(\mathsf{Id} + \sigma(U)) \\ &\quad + H\,(x^k \delta(U) + U\,B - B\,\sigma(U)) \\ &= R\,(\mathsf{Id} + \sigma(U)) - R \bmod x^N = R\,\sigma(U) \bmod x^N. \end{aligned}$$

Using either Lemma 4.10 or Lemma 4.11, $U = 0 \bmod x^{m-k+1}$, so $\sigma(U) = 0 \bmod x^{m-k+1}$. Thus, the latter expression is 0, since $2m-k+1 \geq N$. Finally, since $HU = 0 \bmod x^{m-k+1}$, and $m \geq k$, $H + HU$ remains invertible in $\mathcal{M}_n(\Bbbk[[x]])$. The various matrix products and inversions take a total number of $\mathcal{O}(n^\omega \mathsf{M}(N))$ operations in $\Bbbk$ (using Newton iteration to invert H). Adding the cost of Lemma 4.10, resp. Lemma 4.11, we get the announced complexity. □

We can now prove Proposition 4.9. Correctness is obvious by repeated applications of the previous lemma. The cost $C(N)$ of the computation up to precision N satisfies

$$C(N) = C(m) + \mathcal{O}(n^\omega \mathsf{M}(N) + n^\omega \log n\, N), \quad N > k.$$

Using the super-additivity properties of the function M as in [GG03, Ch. 9], we obtain the claimed complexity.

We can now conclude the proof of Thm. 4.3. In order to solve Equation (4.3), we first determine B and V as in §4.3.3; the cost will be negligible. Then, we use Proposition 4.9 to compute a matrix W that satisfies (4.9) $\bmod x^N$. Given C in $\mathcal{M}_{n,1}(\Bbbk[[x]])$, we next compute $\Gamma = W^{-1}\, C \bmod x^N$. By the previous lemma, we conclude by solving

$$x^k\, \delta(Y) = B\, \sigma(Y) + \Gamma \bmod x^N.$$

Lemma 4.7 gives us generators of the solution space of this equation $\bmod x^N$. If it is inconsistent, we infer that Eq. (4.3) is. Else, from the generators (Y, M) obtained in Lemma 4.7, we deduce that $(WY, WM) \bmod x^N$ is a generator of the solution space of Eq. (4.3) $\bmod x^N$. Since the matrix M has few columns (at most n), the cost of all these computations is dominated by that of Proposition 4.9, as reported in Thm. 4.3.

4.4 Implementation

We implemented the divide-and-conquer and Newton iteration algorithms, as well as a quadratic time algorithm, on top of NTL 5.5.2 [S+90]. In our experiments, the base field is $\Bbbk = \mathbb{Z}/p\mathbb{Z}$, with p a 28 bit prime; the systems were drawn at random. Timings are in seconds, averaged over 50 runs; they are obtained on a single core of a 2 GHz Intel Core 2.

Our implementation uses NTL's built-in `zz_pX` polynomial arithmetic, that is, works with "small" prime fields (of size about 2^{30} over 32 bit machines, and 2^{50} over 64 bits machines). For this data type, NTL's polynomial arithmetic uses a combination of naive, Karatsuba and FFT arithmetic.

There is no built-in NTL type for polynomial matrices, but a simple mechanism to write one. Our polynomial matrix product is naive, of cubic cost. For small sizes such as $n = 2$ or $n = 3$, this is sufficient; for larger n, one should employ improved schemes (such as Waksman's [Wak70], see also [DIS11]) or evaluation-interpolation techniques [BS05].

Our implementation follows the descriptions given above, up to a few optimizations for algorithm NewtonAE (which are all classical in the context of Newton iteration). For instance, the inverse of H should not be recomputed at every step, but simply updated; some products can be computed at a lower precision than it appears (such as $H^{-1}R$, where R is known to have a high valuation).

In Fig. 4.1, we give timings for the scalar case, with $k=1$ and $q \neq 1$. Clearly, the quadratic algorithm is outperformed for almost all values of N; Newton iteration performs better than the divide-and-conquer approach, and both display a sub-quadratic behavior. Fig. 4.2 gives timings when n varies, taking $k=1$ and $q \neq 1$ as before. For larger values of n, the divide-and-conquer approach become much better for this range of values of N, since it avoids costly polynomial matrix multiplication (see Thm. 4.2).

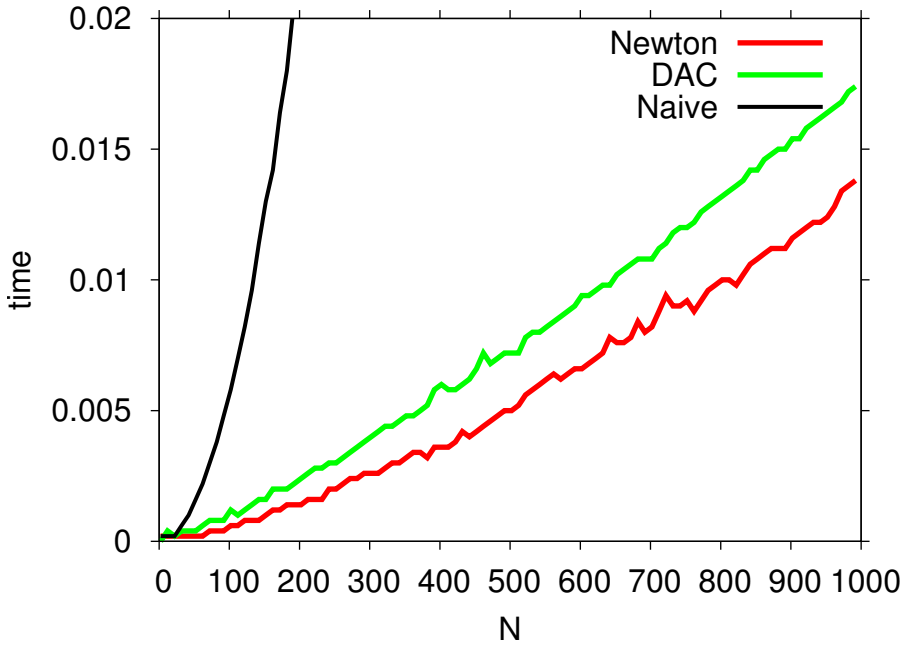

Figure 4.1. Timings with $n=1$, $k=1$, $q \neq 1$

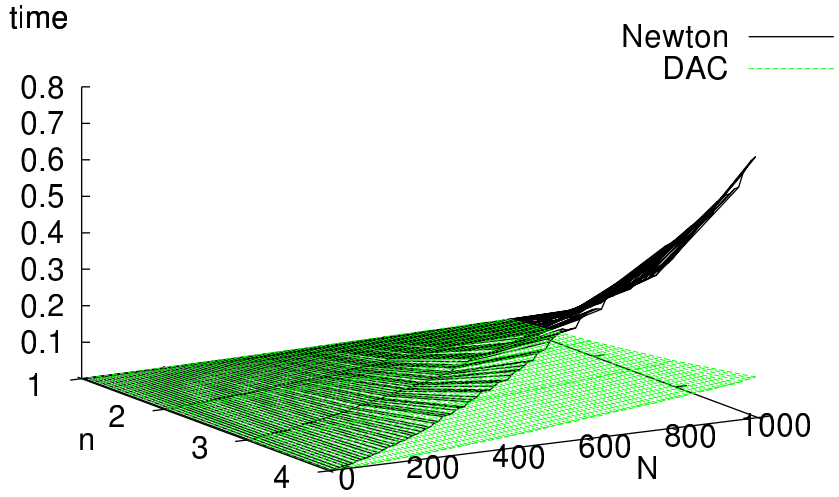

Figure 4.2. Timings with $k=1$, $q \neq 1$

Finally, Table 4.1 gives timings obtained for $k=3$, for larger values of n (in this case, a plot of the results would be less readable, due to the large gap between the divide-and-conquer approach and Newton iteration, in favor of the former); DAC stands for "divide-and-conquer". In all cases, the experimental results confirm to a very good extent the theoretical cost analyses.

Newton		n			
		5	9	13	17
N	50	0.01	0.11	0.32	0.72
	250	0.22	1.2	3.7	8.1
	450	0.50	2.8	8.3	18
	650	0.93	5.1	16	34

DAC		n			
		5	9	13	17
N	50	0.01	0.01	0.02	0.04
	250	0.03	0.07	0.15	0.25
	450	0.06	0.16	0.32	0.52
	650	0.10	0.27	0.53	0.88

Table 4.1. Timings with $k = 3$, $q \neq 1$

Partie III

Algebraic lifting

Chapitre 5

Relaxed p-adic Hensel lifting for algebraic systems

This chapter is the main part of the homonym paper published with J. Berthomieu in the proceedings of *ISSAC'12* [BL12].

In this chapter, we show how to transform algebraic equations into recursive equations. As a consequence, we can use relaxed algorithms to compute the Hensel lifting of a root from the residue ring $R/(p)$ to its p-adic ring R_p. This chapter can be seen as a special and simpler case of lifting of triangular set done in Chapter 6.

We work under the hypothesis of Hensel's lemma, which requires that the derivative at the point we wish to lift is not zero. Our algorithms are worse by a logarithmic factor in the precision compared to Newton iteration. However, the constant factors hidden in the big-O notation are potentially smaller. Moreover, our algorithm's cost is roughly the cost of evaluating the implicit equation by on-line algorithms. This can lead to further savings compared to the cost of off-line methods. For example, consider the multivariate Newton-Hensel operator which performs at each step an evaluation of the implicit equations and an inversion of its evaluated Jacobian matrix. In Theorems 5.11 and 5.14, we manage to save the cost of the inversion of the Jacobian matrix at full precision.

Finally, we implement these algorithms to obtain timings competitive with Newton and even lower on wide ranges of input parameters. As an application, we solve linear systems over the integers and compare to Linbox and IML. We show that we improve the timings for small matrices and big integers.

Our results on the transformation of implicit equations to recursive equations were discovered independently at the same time by [Hoe11]. This paper deals with more general recursive power series defined by algebraic, differential equations or a combination thereof. However, its algorithms have yet to be implemented and only work in characteristic zero. Furthermore, since the carry is not dealt with, the blockwise product as presented in [BHL11, Section 4] cannot be used. This is important because blockwise relaxed algorithms are often an efficient alternative.

5.1 Univariate root lifting

In [BHL11, Section 7], it is shown how to compute the dth root of a p-adic number a in a recursive relaxed way, d being relatively prime to p. In this section, we extend this result to the relaxed lifting of a simple root of any polynomial $P \in R[Y]$. Hensel's lemma ensures that from any modular simple root $y_0 \in R/(p)$ of $\bar{P} \in R/(p)[Y]$, there exists a unique lifted root $y \in R_p$ of P such that $y = y_0 \bmod p$.

From now on, P is a polynomial with coefficients in R and $y \in R_p$ is the unique root of P lifted from the modular simple root $y_0 \in R/(p)$.

Proposition 5.1. *The polynomial*

$$\Phi(Y) := \frac{P'(y_0)\,Y - P(Y)}{P'(y_0)} \in K[Y]$$

allows the computation of y.

Proof. It is clear that if $P(y) = 0$ and $P'(y_0) \neq 0$, then $y = \frac{P'(y_0)\,y - P(y)}{P'(y_0)} = \Phi(y)$. Furthermore, $\Phi'(y_0) = 0$. □

In the following subsections, we will derive some shifted algorithms associated to the recursive equation Φ depending on the representation of P.

5.1.1 Dense polynomials

In this subsection, we fix a polynomial P of degree d given in dense representation, that is as the vector of its coefficients in the monomial basis $(1, Y, \ldots, Y^d)$. To have a shifted algorithm, we need to express $\Phi(Y)$ with a positive shift. Recall, from Definition 2.11, that the shift of $\Phi(Y)$ is 0. In this chapter, for any two p-adics a and b, we denote by $a \cdot b$ their multiplication. If at least one of them has finite precision, we denote by $a\,b$ their multiplication.

Lemma 5.2. *The s.l.p.* $\Gamma\colon Z \mapsto p^2 \times \left(\left(\frac{Z - y_0}{p}\right)^2 \cdot Z^k\right)$ *for* $k \in \mathbb{N} - \{0\}$ *is executable on* y *and* $\mathrm{sh}(\Gamma) = 1$.

Proof. Since $y_0 = y \bmod p$, $\Gamma(y) \in R_p$ and thus Γ is executable on y. Furthermore, the shift $\mathrm{sh}(\Gamma)$ equals $2 + \min\left(\mathrm{sh}\left(\frac{Z - y_0}{p}\right), \mathrm{sh}(Z)\right) = 1$. □

We are now able to derive a shifted algorithm for Φ.

Algorithm - Dense polynomial root lifting
Input: $P \in R[Y]$ with a simple root y_0 in $R/(p)$. **Output:** A shifted algorithm Ψ associated to Φ and y_0.
1. Compute $Q(Y)$ the quotient of $P(Y)$ by $(Y - y_0)^2$
2. Let $\mathrm{sq}(Z)\colon Z \mapsto \left(\frac{Z - y_0}{p}\right)^2$
3. **return** the shifted algorithm Ψ:
$Z \to \frac{-1}{P'(y_0)}\left(P(y_0) - P'(y_0)\,y_0 + p^2 \times (Q(Z) \cdot \mathrm{sq}(Z))\right)$.

Proposition 5.3. *Given a polynomial P of degree d in dense representation and a modular simple root y_0, Algorithm 5.2.1 defines a shifted algorithm Ψ associated to Φ. The precomputation of such an operator involves $\mathcal{O}(d)$ operations in R. If λ is the length of $P'(y_0)$, then we can lift y at precision N in time*

$$(d-1)\,\mathsf{R}(N) + \mathcal{O}(N\,d + N\,\mathsf{R}(\lambda)/\lambda)$$

or equivalently

$$(d-1)\,\mathsf{R}(N) + \mathcal{O}(N\,d) + N \log(\lambda)^{\mathcal{O}(1)}.$$

Proof. First, Ψ is a shifted algorithm for Φ. Indeed since $\mathrm{sh}(P(y_0) - P'(y_0)\,y_0) = +\infty$ and, due to Lemma 5.2, $\mathrm{sh}(p^2 \times (\mathrm{sq}(Z) \cdot Q(Z))) = 1$, we have $\mathrm{sh}(\Psi) = 1$. Also, thanks to Lemma 5.2, we can execute Ψ on y over the R-algebra R_p. Moreover, it is easy to see that $\Phi(Y) = \Psi(Y)$ over the R-algebra $K[Y]$.

The quotient polynomial Q is precomputed in time $\mathcal{O}(d)$ *via* the naïve Euclidean division algorithm. Using Horner scheme to evaluate $Q(Z)$, we have $L^*(\Psi) = d-1$ and we can apply Proposition 2.17. Note that by Proposition 3.6 for $r = 1$, the inversion of $P'(y_0)$ costs $\mathcal{O}(N\,\mathsf{R}(\lambda)/\lambda)$. Finally, the evaluation of Q also involves $\mathcal{O}(d)$ on-line additions which cost $\mathcal{O}(N\,d)$. □

In comparison, Newton iteration lifts y at precision N in time $(3\,d + \mathcal{O}(1))\,\mathsf{I}(N) + \mathcal{O}(d\,N)$ (see [GG03, Theorem 9.25]). Here, the universal constant in the $\mathcal{O}(1)$ corresponds to p-adic inversion and can be taken less than 4. The reminder on Newton iteration can be found in Section 6.3.1.

So the first advantage of our on-line algorithm is that it does asymptotically less on-line multiplications than Newton iteration does off-line multiplications. Also, we can expect better timings from the on-line method for the Hensel lifting of y when the precision N satisfies $\mathsf{R}(N) \leqslant 3\,\mathsf{I}(N)$.

5.1.2 Polynomials as straight-line programs

In [BHL11, Proposition 7.1], the case of the polynomial $P(Y) = Y^d - a$ was studied. Although the general concept of a shifted algorithm was not introduced, an algorithm of multiplicative complexity $\mathcal{O}(L^*(P))$ was given. The shifted algorithm was only present in the implementation in Mathemagix [HLM+02]. We clarify and generalize this approach to any polynomial P given as an s.l.p. and propose a shifted algorithm Ψ whose complexity is linear in $L^*(P)$.

In this subsection, we fix a polynomial P given as an s.l.p. Γ with L operations in $\Omega := \{+, -, \cdot\} \cup R \cup R^c$ and multiplicative complexity $L^* := L^*(P)$, and a modular simple root $y_0 \in R/(p)$ of P. Then, we define the polynomials $T_P(Y) := P(y_0) + P'(y_0)\,(Y - y_0)$ and $E_P(Y) := P(Y) - T_P(Y)$.

Definition 5.4. *We define recursively a vector $\tau \in R^2$ and an s.l.p. ε with operations in $\Omega' := \{+, -, \cdot, p^i \times \_, \_/p^i\} \cup R \cup R^c$. Initially, $\varepsilon^0 := 0$ and $\tau^0 := (y_0, 1)$. Then, we define ε^i and τ^i recursively on i with $1 \leqslant i \leqslant L$ by:*

- *if $\Gamma_i = (a^c;)$, then $\varepsilon^i := 0$, $\tau^i := (a, 0)$;*
- *if $\Gamma_i = (a \times \_; u)$, then $\varepsilon^i := a \times \varepsilon^u$, $\tau^i := a\,\tau^u$;*

- *if* $\Gamma_i = (\pm; u, v)$, *then* $\varepsilon^i := \varepsilon^u \pm \varepsilon^v$, $\tau^i := \tau^u \pm \tau^v$;
- *if* $\Gamma_i = (\cdot; u, v)$ *and we denote by* $\tau^u = (a, A)$, $\tau^v = (b, B)$, *then* $\tau^i = (a\,b,\ a\,B + b\,A)$ *and* ε^i *equals*

$$\varepsilon^u \cdot \varepsilon^v + p \times (((A \times \varepsilon^v + B \times \varepsilon^u)/p) \cdot (Z - y_0)) + (a \times \varepsilon^v + b \times \varepsilon^u) + p^2 \times ((A\,B) \times ((Z - y_0)/p)^2). \tag{5.1}$$

Recall that multiplications denoted by $\cdot$ *are the ones between p-adics. Finally, we set* $\varepsilon_P := \varepsilon^L$ *and* $\tau_P := \tau^L$ *where* L *is the number of instructions in the s.l.p.* P.

Lemma 5.5. *The s.l.p.* ε_P *is a shifted algorithm for* E_P *and* y_0. *Its multiplicative complexity is bounded by* $2\ L^* + 1$. *Also,* τ_P *is the vector of coefficients of the polynomial* T_P *in the basis* $(1, (Y - y_0))$.

Proof. Let us call P_i the ith result of the s.l.p. P on the input Y over $R[Y]$, with $0 \leqslant i \leqslant L$. We denote by $E^i := E_{P_i}$ and $T^i := T_{P_i}$ for all $0 \leqslant i \leqslant L$. Let us prove recursively that ε^i is a shifted algorithm for E^i and y_0, and that τ^i is the vector of coefficients of T^i in the basis $(1, (Y - y_0))$.

For the initial step $i = 0$, we have $P_0 = Y$ and we verify that $E^0(Y) = \varepsilon^0(Y) = 0$ and $T^0(Y) = y_0 + (Y - y_0)$. The s.l.p. ε^0 is executable on y over R_p and its shift is $+\infty$.

Now we prove the result recursively for $i > 0$. We detail the case when $\Gamma_i = (\cdot; u, v)$, the others cases being straightforward. Equation (5.1) corresponds to the last equation of

$$\begin{array}{rcl} P_i & = & P_u\,P_v \\ \Leftrightarrow\ E^i & = & (E^u + T^u)\,(E^v + T^v) - T^i \\ \Leftrightarrow\ E^i & = & E^u\,E^v + [T^v\,E^u + T^u\,E^v] + (T^u\,T^v - T^i) \\ \Leftrightarrow\ E^i & = & E^u\,E^v + [(P'_u(y_0)\,E^v + P'_v(y_0)\,E^u)\,(Y - y_0) + (P_u(y_0)\,E^v + P_v(y_0)\,E^u)] \\ & & + P'_u(y_0)\,P'_v(y_0)\,(Y - y_0)^2. \end{array}$$

Also $\tau^i = (P_u(y_0)\,P_v(y_0), P_u(y_0)\,P'_v(y_0) + P_v(y_0)\,P'_u(y_0))$. The s.l.p. ε^i is executable on y over R_p because, for all $j < i$, $\mathrm{sh}(\varepsilon_j) > 0$ implies that $(A\,\varepsilon^v(y) + B\,\varepsilon^u(y))/p \in R_p$. Concerning the shifts, since $\mathrm{sh}(\varepsilon_u), \mathrm{sh}(\varepsilon_v) > 0$, we can check that every operand in Equation (5.1) has a positive shift. So $\mathrm{sh}(\varepsilon^i) > 0$. Then, take $i = r$ to conclude the proof.

Concerning multiplicative complexity, we slightly change ε^0 such that it computes once and for all $((Y - y_0)/p)^2$ before returning zero. Then, for all multiplication instructions $\cdot$ in the s.l.p. P, the s.l.p. ε_P adds two multiplications $\cdot$ between p-adics (see Equation (5.1)). So $L^*(\varepsilon_P) = 2\,L^* + 1$. □

Proposition 5.6. *Let* P *be a univariate polynomial over* R_p *given as an s.l.p. whose multiplicative complexity is* L^*. *Then, the following algorithm*

$$\Psi\colon Z \mapsto \frac{-P(y_0) + P'(y_0)\,y_0 - \varepsilon_P(Z)}{P'(y_0)}$$

is a shifted algorithm associated to Φ *and* y_0 *whose multiplicative complexity is* $2\,L^* + 1$.

Proof. We have $\Phi(Y) = \Psi(Y)$ over the algebra $K[Y]$ because $\Phi(Y) = (-P(y_0) + P'(y_0)\, y_0 + E_P(Y))/P'(y_0)$. Because of Lemma 5.5 and $\nu_p(P'(y_0)) = 0$, the s.l.p. Ψ is executable on y over R_p and its shift is positive. We conclude with $L^*(\Psi) = L^*(\varepsilon_P) = 2\,L^* + 1$ as the on-line division by $P'(y_0)$ does not require any multiplication between full precision p-adics (see Chapter 3). □

Remark 5.7. By adding the square operation $\_^2$ to the set of operations Ω of P, we can save a few multiplications. In Definition 5.4, if $\Gamma_i = (\_^2; u)$ and $\tau^u = (a, A)$, then we define ε^i by $\varepsilon^u \cdot (\varepsilon^u + 2 \times (a + A \times (Z - y_0))) + p^2 \times (A^2 \times ((Z - y_0)/p)^2)$. Thereby, we reduce the multiplicative complexity of ε_P and Ψ by the number of square operations in P.

Theorem 5.8. *Let $P \in R[Y]$ and $y_0 \in R/(p)$ be such that $P(y_0) = 0 \bmod p$ and $P'(y_0) \neq 0 \bmod p$. Denote by $y \in R_p$ the unique solution of P lifted from y_0. Assume that P is given as an s.l.p. with operations in $\Omega := \{+, -, \cdot\} \cup R \cup R^c$ whose multiplicative complexity is L^*. Let λ be a bound on the length of all elements $P_i(y_0)$ in the result sequence of the evaluation of P at y_0 and on all $r \in R$ such that $r \times \_$ is an operation of the s.l.p. P.*

Then, we can lift y up to precision N in time

$$(2\,L^* + 1)\,\mathsf{R}(N) + \mathcal{O}(N\,L\,\mathsf{R}(\lambda)/\lambda),$$

that is

$$(2\,L^* + 1)\,\mathsf{R}(N) + N\,L \log(\lambda)^{\mathcal{O}(1)}.$$

Proof. By Propositions 5.1 and 5.6, y can be computed as a recursive p-adic number with the shifted algorithm Ψ. Proposition 2.17 gives that the cost of lifting y up to precision N is the cost of evaluating $\Psi(y)$ at precision N. This evaluation requires $(2\,L^* + 1)$ on-line multiplications, $\mathcal{O}(L)$ additions, $\mathcal{O}(L)$ multiplications between p-adics with one operand of finite length $\mathcal{O}(\lambda)$ (coming either from operations $r \times \_$ or $\cdot$ in P) and a division by $P'(y_0)$ for a total cost of

$$(2\,L^* + 1)\,\mathsf{R}(N) + \mathcal{O}(N\,L + N\,L\,\mathsf{R}(\lambda)/\lambda + N\,\mathsf{R}(\lambda)/\lambda). \qquad \square$$

In this case, Newton iteration costs $(4\,L^* + \mathcal{O}(1))\,\mathsf{I}(N) + \mathcal{O}(L\,N)$. To prove this claim, we have to show that the evaluation of P at precision N costs $L^*\,\mathsf{I}(N) + \mathcal{O}(L\,N)$, and that the evaluation of P' at precision $N/2$ costs $2\,L^*\,\mathsf{I}(N/2) + \mathcal{O}(L\,N)$. One way to compute $(P(y), P'(y))$ is to evaluate P at $y + \varepsilon$ in the ring of tangent numbers $R_p[\varepsilon]/\varepsilon^2$. Then $P(y + \varepsilon) = P(y) + \varepsilon\,P'(y)$. Note that

$$\begin{aligned} (a + b\,\varepsilon) + (c + d\,\varepsilon) &= (a + c) + (b + d)\,\varepsilon \\ (a + b\,\varepsilon)\,(c + d\,\varepsilon) &= a\,c + (b\,c + a\,d)\,\varepsilon \end{aligned}$$

in $R_p \oplus R_p\,\varepsilon = R_p[\varepsilon]/\varepsilon^2$. Consequently a multiplication in $R_p[\varepsilon]/\varepsilon^2$ costs 3 multiplications in R_p. But because we want the coefficient in ε at precision only $N/2$, we require b and d at precision $N/2$. Therefore by evaluating P at $y + \varepsilon$ in $R/(p^N) \oplus R/(p^{N/2})\ \varepsilon$, we obtain $P(y)$ at precision N and $P'(y)$ at precision $N/2$ in time $2\,L^*\,\mathsf{I}(N/2) + L^*\,\mathsf{I}(N) + \mathcal{O}(L\,N)$. The inversion of $P'(y)$ costs $\mathcal{O}(\mathsf{I}(N))$.

Therefore the last step of Newton iteration costs $(2\,L^* + \mathcal{O}(1))\,\mathsf{I}(N) + \mathcal{O}(L\,N)$. Finally, the whole Newton iteration involves the steps $N, N/2, N/4, \ldots$ for a total cost of $(4\,L^* + \mathcal{O}(1))\,\mathsf{I}(N) + \mathcal{O}(L\,N)$.

Remark 5.9. We can improve the bound on the multiplicative complexity when the polynomial has a significant part with positive valuation. Indeed suppose that the polynomial P is given as $P(Y) = \alpha(Y) + p\,\beta(Y)$ with α and β two s.l.p.'s. Then the part $p\,\beta(Y)$ is already shifted. In this case, set $\tilde{\varepsilon}_P := \varepsilon_\alpha + p\,\beta$ so that

$$\Psi\colon Z \mapsto \frac{-\alpha(y_0) + \alpha'(y_0)\,y_0 - \tilde{\varepsilon}_P(Z)}{\alpha'(y_0)}$$

is a shifted algorithm for P with multiplicative complexity $2\,L^*(\alpha) + L^*(\beta) + 1$.

5.2 Multivariate root lifting

In this section, we lift a p-adic root $\boldsymbol{y} \in R_p^r$ of a polynomial system $\boldsymbol{P} = (P_1, \ldots, P_r) \in R[\boldsymbol{Y}]^r = R[Y_1, \ldots, Y_r]^r$ in a relaxed recursive way. We make the assumption that $\boldsymbol{y}_0 = (y_{1,0}, \ldots, y_{r,0}) \in (R/(p))^r$ is a regular modular root of $\boldsymbol{P}$, *i.e.* its Jacobian matrix $\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)$ is invertible in $\mathcal{M}_r\,(R/(p))$. The Newton-Hensel operator ensures both the existence and the uniqueness of $\boldsymbol{y} \in R_p^r$ such that $\boldsymbol{P}(\boldsymbol{y}) = 0$ and $\boldsymbol{y}_0 = \boldsymbol{y} \bmod p$. From now on, $\boldsymbol{P}$ is a polynomial system with coefficients in R and $\boldsymbol{y} \in R_p^r$ is the unique root of $\boldsymbol{P}$ lifted from the modular regular root $\boldsymbol{y}_0 \in (R/(p))^r$.

Proposition 5.10. *The polynomial system*

$$\boldsymbol{\Phi}(\boldsymbol{Y}) := \mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)^{-1}(\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)\,\boldsymbol{Y} - \boldsymbol{P}(\boldsymbol{Y})) \in K[\boldsymbol{Y}]^r$$

allows the computation of $\boldsymbol{y}$.

Proof. We adapt the proof of Proposition 5.1. Since $\mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{y}_0) = \boldsymbol{0}$, $\boldsymbol{\Phi}$ allows the computation of $\boldsymbol{y}$. □

As in the univariate case, we have to introduce a positive shift in $\boldsymbol{\Phi}$. In the following, we present how to do so depending on the representation of $\boldsymbol{P}$.

5.2.1 Dense algebraic systems

In this subsection, we assume that the algebraic system $\boldsymbol{P}$ is given in dense representation. We assume that $d \geqslant 2$, where $d := \max_{1 \leqslant i,j \leqslant r}(\deg_{X_j}(P_i)) + 1$, so that the dense size of $\boldsymbol{P}$ is bounded by $r\,d^r$.

As in the univariate case, the shift of $\boldsymbol{\Phi}(\boldsymbol{Y})$ is 0. We adapt Lemma 5.2 and Proposition 5.3 to the multivariate polynomial case as follows. For $1 \leqslant j \leqslant k \leqslant r$, let $\boldsymbol{Q}^{(j,k)}$ be polynomial systems such that $\boldsymbol{P}(\boldsymbol{Y})$ equals

$$\boldsymbol{P}(\boldsymbol{y}_0) + \mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)\,\boldsymbol{Y} + \sum_{1 \leqslant j \leqslant k \leqslant r} \boldsymbol{Q}^{(j,k)}(\boldsymbol{Y})\,(Y_j - y_{j,0})\,(Y_k - y_{k,0}).$$

Algorithm - Dense polynomial system root lifting

Input: $\boldsymbol{P} \in R[\boldsymbol{Y}]^r$ with a regular root $\boldsymbol{y}_0$ in $(R/(p))^r$.
Output: A shifted algorithm $\boldsymbol{\Psi}$ associated to $\boldsymbol{\Phi}$ and $\boldsymbol{y}_0$.

1. For $1 \leqslant j \leqslant k \leqslant r$, compute a $\boldsymbol{Q}^{(j,k)}(\boldsymbol{Y})$ from $\boldsymbol{P}(\boldsymbol{Y})$
2. For $1 \leqslant j \leqslant k \leqslant r$, let $\mathrm{pr}_{j,k}(\boldsymbol{Z}) := \left(\frac{Z_j - y_{j,0}}{p}\right)\left(\frac{Z_k - y_{k,0}}{p}\right)$
3. Let $\boldsymbol{\Psi}_1 \colon \boldsymbol{Z} \mapsto \sum_{1 \leqslant j \leqslant k \leqslant r} \boldsymbol{Q}^{(j,k)}(\boldsymbol{Z}) \cdot \mathrm{pr}_{j,k}(\boldsymbol{Z})$
4. **return** the shifted algorithm

   $\boldsymbol{\Psi} \colon \boldsymbol{Z} \mapsto -\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)^{-1}(\boldsymbol{P}(\boldsymbol{y}_0) - \mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)\, \boldsymbol{y}_0 + p^2 \times \boldsymbol{\Psi}_1)$.

Theorem 5.11. *Let $\boldsymbol{P} = (P_1, \ldots, P_r)$ be a polynomial system in $R[\boldsymbol{Y}]^r$ in dense representation, satisfying $d \geqslant 3$ where $d := \max_{1 \leqslant i,j \leqslant r} (\deg_{X_j}(P_i)) + 1$, and let $\boldsymbol{y}_0$ be an approximate zero of $\boldsymbol{P}$. Let λ be a bound on the length of the polynomial coefficients of $\boldsymbol{P}$ and on the entries of $\mathrm{Jac}_{\boldsymbol{P}}(y_0)$.*

Then Algorithm 5.2.1 outputs a shifted algorithm $\boldsymbol{\Psi}$ associated to $\boldsymbol{\Phi}$ and $\boldsymbol{y}_0$. The precomputation in $\boldsymbol{\Psi}$ costs $\mathcal{O}(r\, d^r)$ operations in R, while computing $\boldsymbol{y}$ up to precision N costs

$$d^r\, \mathsf{R}(N) + \mathcal{O}(N\, [r\, d^r\, \mathsf{R}(\lambda)/\lambda + \mathsf{MMR}(r, 1, \lambda)/\lambda] + r^\omega),$$

that is

$$d^r\, \mathsf{R}(N) + N\, r\, d^r \log(\lambda)^{\mathcal{O}(1)} + \mathcal{O}(r^\omega).$$

Proof. First, for $j \leqslant r$, we perform the Euclidean division of $\boldsymbol{P}$ by $(Y_j - y_{j,0})^2$ to reduce the degree in each variable. The naïve algorithm does the first division in time $\mathcal{O}(r\, d^r)$. Then the second division costs $\mathcal{O}(r\, 2\, d^{r-1})$ because we reduce a polynomial with less monomials. The third $\mathcal{O}(r\, 2^2\, d^{r-2})$ and so on. At the end, all these divisions are done in time $\mathcal{O}(r\, d^r)$. Then, for each P_i, it remains a polynomial with partial degree at most 1 in each variable. Necessary divisions by $(Y_j - y_{j,0})\,(Y_k - y_{k,0})$ are given by the presence of a multiple of $Y_j\, Y_k$, which gives rise to a cost of $\mathcal{O}(2^r) = o(r\, d^r)$. Finally, the entries of the Jacobian matrix $\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)$ are obtained as the coefficients in $(Y_j - y_{j,0})$ of the resulting polynomial and $\boldsymbol{P}(\boldsymbol{y}_0)$ as the constant coefficient. The multiplication $\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)\, \boldsymbol{y}_0$ takes $\mathcal{O}(r^2) = o(r\, d^r)$ operations in R.

Next, we have to evaluate $\boldsymbol{\Psi}_1$ at $\boldsymbol{y}$. We start by computing the evaluation at $\boldsymbol{y}$ of all the monomials appearing in $\boldsymbol{\Psi}_1$. There are at most d^r monomials. Since each monomial, except 1, is obtained as the product of another monomial by one Z_j with $1 \leqslant j \leqslant r$, all these evaluations take d^r on-line multiplications.

Then, for each component of the vector $\boldsymbol{\Psi}_1$, we multiply the monomials by the corresponding polynomial coefficient in R and had these terms together. These coefficients have length λ, hence a cost $\mathcal{O}(N\, r\, d^r\, \mathsf{R}(\lambda)/\lambda + N\, r\, d^r)$.

Finally, we have to multiply this by the inverse of the Jacobian of $\boldsymbol{P}$ at $\boldsymbol{y}_0$, which is a matrix with coefficients in R of length λ. By Proposition 3.6, and since we only lift a single root, it can be done at precision N in time $\mathcal{O}(N\, \mathsf{MMR}(r, 1, \lambda)/\lambda + r^\omega)$. We conclude with the relation $\mathsf{MMR}(r, 1, \lambda) = \tilde{\mathcal{O}}(r^2 \log(\lambda)^{\mathcal{O}(1)})$. □

Once again, we compare with Newton iteration which performs at each step an evaluation of the polynomial equations and of their Jacobian matrix, and an inversion of its evaluated Jacobian matrix. This would amount to a cost $\mathcal{O}((r\,d^r + r^\omega)\,\mathsf{I}(N))$, since both the evaluations cost $\mathcal{O}(r\,d^r)$ arithmetic operations on p-adics. The latter theorem shows that we manage to save the cost of the inversion of the Jacobian matrix at full precision with on-line algorithms.

This latter advantage is meaningful when the cost of evaluation of the system is lower than the cost of linear algebra. Therefore we adapt our on-line approach to polynomials given as straight-line programs.

5.2.2 Algebraic systems as s.l.p.'s

In this subsection, we assume that the algebraic system $\boldsymbol{P}$ is given as an s.l.p. We keep basically the same notations as in Section 5.1.2. Given an algebraic system $\boldsymbol{P}$, we define $\boldsymbol{T_P}(\boldsymbol{Y}) := \boldsymbol{P}(\boldsymbol{y}_0) + \mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)\,(\boldsymbol{Y} - \boldsymbol{y}_0)$ and $\boldsymbol{E_P}(\boldsymbol{Y}) := \boldsymbol{P}(\boldsymbol{Y}) - \boldsymbol{T_P}(\boldsymbol{Y})$. We adapt Definition 5.4 so that we may define τ and ε for multivariate polynomials.

Definition 5.12. *We define recursively $\tau_i \in R \times R_p$, $\varepsilon_i \in R_p$ for $1 \leqslant i \leqslant r$ with operations in $\Omega' := \{+, -, \cdot, p^j \times \_, \_/p^j\} \cup R \cup R^c$.*

Initialize $\varepsilon_i^{-r+j} := 0$, $\tau_i^{-r+j} := (y_{j,0},\ y_j - y_{j,0})$ for all $1 \leqslant j \leqslant r$. Then for $1 \leqslant j \leqslant L_i$ where L_i is the number of instructions in the s.l.p. P_i, we define ε_i^j and τ_i^j recursively on j by formulas similar to Definition 5.4. Let us detail the changes when $\Gamma_j = (\cdot; u, v)$:

Let $\tau_i^u = (a, A)$ and $\tau_i^v = (b, B)$, then define τ_i^j by $(a\,b, a \times B + b \times A)$ and ε_i^j by

$$p \times \left((a + A + \varepsilon_i^u) \cdot \frac{\varepsilon_i^v}{p} + (b + B) \cdot \frac{\varepsilon_i^u}{p} \right) + p^2 \times \left(\frac{A}{p} \cdot \frac{B}{p} \right).$$

As before, we set $\varepsilon_{P_i} := \varepsilon_i^{L_i}$ and $\tau_{P_i} := \tau_i^{L_i}$.

Lemma 5.13. *If $\tau_{P_i} = (a,\ A)$ then $a = P_i(\boldsymbol{y}_0)$ and $A = \mathrm{Jac}_{P_i}(\boldsymbol{y}_0)\ (\boldsymbol{Y} - \boldsymbol{y}_0) \in R_p$. Besides, $\boldsymbol{\varepsilon_P} := (\varepsilon_{P_1}, \ldots, \varepsilon_{P_r})$ is a shifted algorithm for $\boldsymbol{E_P}$ and $\boldsymbol{y}_0$ whose complexity is $3\,L^*$.*

Proof. Following the proof of Lemma 5.5, the first assertion is clear, as is the fact that $\boldsymbol{\varepsilon_P}$ is a shifted algorithm for $\boldsymbol{E_P}$ and $\boldsymbol{y}_0$. Finally, for all instructions $\cdot$ in the s.l.p. P_i, ε_{P_i} adds three multiplications between p-adics (see operations $\cdot$ in formulas above). So $L^*(\boldsymbol{\varepsilon_P}) = 3\,L^*$. □

Theorem 5.14. *Let $\boldsymbol{P}$ be a polynomial system of r polynomials in r variables over R, given as an s.l.p. such that its multiplicative complexity is L^*. Let $\boldsymbol{y}_0 \in (R/(p))^r$ be such that $\boldsymbol{P}(\boldsymbol{y}_0) = 0 \bmod p$ and $\det(\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)) \neq 0 \bmod p$. Denote by $\boldsymbol{y}$ the unique solution of $\boldsymbol{P}$ lifted from $\boldsymbol{y}_0$. Let λ be a bound on the length of all $r \in R$ such that $r \times \_$ is an operation of $\boldsymbol{P}$, all elements $P_i(\boldsymbol{y}_0)$ in the result sequence of the evaluation of $\boldsymbol{P}$ at $\boldsymbol{y}_0$ and all entries of $\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)$.*

Then, the algorithm

$$\boldsymbol{\Psi}\colon \boldsymbol{Z} \mapsto \mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)^{-1}(-\boldsymbol{P}(\boldsymbol{y}_0) + \mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)\,\boldsymbol{y}_0 - \boldsymbol{\varepsilon}_{\boldsymbol{P}}(\boldsymbol{Z}))$$

is a shifted algorithm associated to $\boldsymbol{\Phi}$ *and* $\boldsymbol{y}_0$*. This algorithm requires a precomputation of* $\mathcal{O}(r\,L + r^2)$ *operations in* R*. Then, one can compute* $\boldsymbol{y}$ *to precision* N *in time*

$$3\,L^*\,\mathsf{R}(N) + \mathcal{O}(N\,[L\,\mathsf{R}(\lambda)/\lambda + \mathsf{MMR}(r,1,\lambda)/\lambda] + r^\omega),$$

or equivalently,

$$3\,L^*\,\mathsf{R}(N) + N\,(L + r^2 \log(r)^{\mathcal{O}(1)}) \log(\lambda)^{\mathcal{O}(1)} + \mathcal{O}(r^\omega).$$

Proof. Similarly to Proposition 5.6, $\boldsymbol{\Psi}$ is a shifted algorithm. In terms of operations in R, the evaluation of $\boldsymbol{P}(\boldsymbol{y}_0)$ and $\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)$ costs $\mathcal{O}(r\,L)$ operations by [BS83], and $\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)\ \boldsymbol{y}_0$ requires $\mathcal{O}(r^2)$ more operations. By Lemma 5.13, the evaluation of $\boldsymbol{\varepsilon}_{\boldsymbol{P}}(\boldsymbol{y})$ cost $3\,L^*$ on-line multiplications, $\mathcal{O}(L)$ on-line additions, $\mathcal{O}(L)$ multiplications between p-adics with one operand of finite length $\mathcal{O}(\lambda)$ (coming either from operations $r \times \_$ or $\cdot$ in $\boldsymbol{P}$) and a division by $\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)$ for a total cost of

$$3\,L^*\,\mathsf{R}(N) + \mathcal{O}(N\,L + N\,L\,\mathsf{R}(\lambda)/\lambda + r^\omega + N\,\mathsf{MMR}(r,1,\lambda)/\lambda). \qquad \square$$

In this case, Newton iteration costs $\mathcal{O}(r\,L^* + r^\omega)\,\mathsf{I}(N) + \mathcal{O}(N\,L)$. Hence our on-line approach is particularly well-suited to systems that can be evaluated cheaply, e.g. sparse polynomial systems.

5.3 Implementation and Timings

In this section, we display computation times in milliseconds for the univariate polynomial root lifting and for the computation of the product of the inverse of a matrix with a vector or with another square matrix. Timings are measured using one core of an Intel Xeon X5650 at 2.67 GHz running Linux, Gmp 5.0.2 [G+91] and setting $p = 536871001$ a 30 bit prime number.

Our implementation is available in the files whose names begin with `series_carry` or `p_adic` in the C++ library algebramix of Mathemagix.

In the following tables, the first line, "Newton" corresponds to the classical off-line Newton iteration [GG03, Algorithm 9.2]. The second line "Relaxed" corresponds to our best variant. The last line gives a few details about which variant is used. We make use of the naive variant "N" and the relaxed variant "R". These variants differ only by the on-line multiplication algorithm used in Algorithm `OnlineEvaluationStep` inside Algorithm `OnlineRecursivePadic` to compute the recursive p-adics (see Section 2.2.2). The naive variant calls Algorithm `LazyMulStep` of Section 1.1.1.3, whereas the relaxed variant calls Algorithm `RelaxedProductStep` of Section 1.1.3.4. In fact, since we work on p-adic integers, the relaxed version uses an implementation of Algorithm $\texttt{Binary\_Mul\_Padic}_p$ from [BHL11, Section 3.2], which is a p-adic integer variant of Algorithm `RelaxedProductStep`.

Furthermore, when the precision is high, we make use of blocks of size 32 or 1024. That means, that at first, we compute the solution f up to precision 32 as $F_0 = f_0 + \cdots + f_{31}\, p^{31}$ with the variant "N". Then, we say that our solution can be seen as a p^{32}-adic integer $F = F_0 + \cdots + F_n\, p^{32n} + \cdots$ and the algorithm runs with F_0 as the initial condition. Then, each F_n is decomposed in base p to retrieve $f_{32n}, \ldots, f_{32n+31}$. Although it is competitive, the initialization of F can be quite expensive. "BN" means that F is computed with the variant "N", while "BR" means it is with the variant "R". Finally, if the precision is high enough, one may want to compute F with blocks of size 32, and therefore f with blocks of size 1024. "B^{2}N" (resp. "B^{2}R") means that f and F are computed up to precision 32 with the variant "N" and then, the p^{1024}-adic solution is computed with the variant "N" (resp. "R").

Polynomial root This table corresponds to the lifting of a regular root from $\mathbb{F}_p$ to $\mathbb{Z}_p$ at precision N as in Section 5.1.1.

N	512	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}
Newton	17	48	140	380	1000	2500	5900
Relaxed	120	140	240	600	1600	4200	11000
Variant	R	BN	BN	BR	BR	BR	BR

Table 5.1. Dense polynomial of degree 127

In this table, the timings of "Newton" are always better than "Relaxed". However, if the unknown required precision is slightly above a power of 2, e.g. $2^\ell + 1$, then one needs to compute at precision $2^{\ell+1}$ with Newton algorithms. Whereas relaxed algorithms increase the precision one by one. So the timings of "Relaxed" are better on significant ranges after powers of 2. Notice that this remark is only valid when the required precision N is not known in advance. Otherwise, we can adapt Newton's iteration to end with precision N or $N + 1$.

Acknowledgments

We would like to thank J. VAN DER HOEVEN, M. GIUSTI, G. LECERF, M. MEZZAROBBA and É. SCHOST for their helpful comments and remarks. For their help with LINBOX, we thank B. BOYER and J.-G. DUMAS.

This work has been partly supported by the DIGITEO 2009-36HD grant of the Région Île-de-France, and by the French ANR-09-JCJC-0098-01 MAGIX project.

Chapitre 6
Relaxed lifting of triangular sets

In this chapter, we present a new lifting algorithm for triangular sets over p-adics. Our contribution is to give, for any p-adic triangular set, a shifted algorithm of which the triangular set is a fixed point. Then we can apply the recursive p-adic framework and deduce a relaxed lifting algorithm for this triangular set.

We compare our algorithm with the adaptation of the Newton-Hensel operator for triangular sets of [GLS01, HMW01, Sch02]. Our algorithm always improves the asymptotic cost in the precision for the special case of univariate representations. The general situation is more contrasted.

Finally we implement these algorithms in the C++ library Algebramix of Mathemagix [HLM+02] for the special case of univariate representations. Our new relaxed algorithm compares favorably on the examples. We mention that our on-line algorithm is currently connected to Kronecker inside Mathemagix with the help of G. Lecerf.

This chapter contains work in progress.

6.1 Introduction

6.1.1 Notations

Throughout this chapter, we use the notions and notations of Chapter 1, Section 1.1. In particular, we use the ring of p-adics R_p with its assumption on the length function λ and its complexity model. We recall that $\mathsf{I}(n)$ and $\mathsf{R}(n)$ denotes the cost of multiplying two p-adics of length n by respectively an off-line and an on-line algorithm.

In this chapter, we choose to denote elements by small letters, e.g. $a \in R_p$, vectors by bold fonts, e.g. $\boldsymbol{a} \in (R_p)^n$, and matrices by capital letters, e.g. $A \in \mathcal{M}_n(R_p)$. We denote by $\boldsymbol{v}_1 \cdot \boldsymbol{v}_2$ the inner product between two vectors and $c \times \boldsymbol{v}$ the coefficientwise product of a scalar c by a vector $\boldsymbol{v}$.

Let $\mathsf{M}(d_1, ..., d_n)$ denote the cost of multiplication of dense multivariate polynomials $P \in R[X_1, ..., X_n]$ satisfying $\deg_{X_i}(P) \leqslant d_i$ for all $1 \leqslant i \leqslant n$. By Kronecker substitution, we get that $\mathsf{M}(d_1, ..., d_n) = \mathcal{O}(\mathsf{M}(2^n\, d_1 \cdots d_n))$. We point to Chapter 1, Section 1.2 for details on the cost function M of polynomial multiplication. We denote by $\langle P_1, ..., P_k \rangle$ the ideal spanned by $P_1, ..., P_k \in R[X_1, ..., X_n]$.

Let us introduce the notion of univariate representation of a zero-dimensional ideal $\mathcal{I} \subseteq R[X_1, ..., X_n]$ for any ring R. An element P of $A := R[X_1, ..., X_n]/\mathcal{I}$ will be called *primitive* if the R-algebra $R[P]$ spanned by P is equal to A itself. If Λ is a primitive linear form in A, a *univariate representation* of A consists of polynomials $\mathfrak{P} = (Q, S_1, ..., S_n)$ in $R[T]$ with $\deg(S_i) < \deg(Q)$ such that we have a R-algebra isomorphism

$$\begin{array}{ccc} A = R[X_1, ..., X_n]/\mathcal{I} & \to & R[T]/(Q) \\ X_1, ..., X_n & \mapsto & S_1, ..., S_n \\ \Lambda & \mapsto & T. \end{array}$$

The oldest trace of this representation is to be found in [Kro82] and a few years later in [Kön03]. A good summary of their work can be found in [Mac16]. The shape lemma [GM89] states the existence of such a representation for a generic linear form Λ of a zero-dimensional ideal. Different algorithms compute this representation, from a geometric resolution [GHMP97, GHH+97, GLS01, HMW01] or using a Gröbner basis [Rou99].

When using univariate representations, the elements of $A \simeq R[T]/(Q)$ are then represented as univariate polynomials of degree less than $d := \deg(Q)$. Then, multiplication in A costs $\mathcal{O}(\mathsf{M}(d))$.

A *triangular set* is a set of n polynomials $\boldsymbol{t} = (t_1, ..., t_n) \subseteq R[X_1, ..., X_n]$ such that t_i is in $R[X_1, ..., X_i]$, monic and reduced with respect to $(t_1, ..., t_{i-1})$. The notion of triangular set comes from [Rit66] in the context of differential algebra. Many similar notions were introduced afterwards [Wu84, Laz91, Kal93, ALMM99]. Although all these notions do not coincide in general, they are the same for zero-dimensional ideals.

As it turns out, univariate representations can be seen as a special case of triangular sets. Indeed, with the notations above, the family $(Q(T), X_1 - S_1(T), ..., X_n - S_n(T))$ is a triangular set in the algebra $R[T, X_1, ..., X_n]$.

For any triangular set $\boldsymbol{t}$ in $R[X_1, ..., X_n]$, we define the number e of *essential variables* by $e := \#\{i | d_i > 1\}$ where $d_i := \deg_{X_i}(t_i)$. If r is a reduced normal form modulo $\boldsymbol{t}$, then r is written on e variables, that is $r \in R[X_j]_{j \in \{i | d_i > 1\}}$. Only those variables play a true role in the quotient algebra $A := R[X_1, ..., X_n]/\langle \boldsymbol{t} \rangle$. We define $\mathsf{Rem}(d_1, ..., d_e)$ to be the cost of reducing polynomials $P \in R[X_1, ..., X_n]$ satisfying $\deg_{X_i}(P) \leqslant 2(d_i - 1)$ modulo $\boldsymbol{t}$. As it turns out, the cost of arithmetic operations in the quotient algebra A is $\mathcal{O}(\mathsf{Rem}(d_1, ..., d_e))$ (see Section 6.2). The number e of *essential variables* plays an important role because $\mathsf{Rem}(d_1, ..., d_e)$ is exponential in e.

As in Chapter 3, we denote by ω the exponent of linear algebra on fields, so that we can multiply and invert matrices in $\mathcal{M}_{n \times n}(R)$ in $\mathcal{O}(n^\omega)$ arithmetic operations. We will also need to invert matrices over rings that are not fields, e.g. in quotients of polynomial ring $R[T]/(Q)$. We denote by $\mathcal{O}(n^\Omega)$ the arithmetic complexity of the elementary operations on $n \times n$ matrices over any commutative ring: addition, multiplication, determinant and adjoint matrix. In fact, Ω can be taken less than 2.70 [Ber84, Kal92, KV04]. For the special case of matrices over $R_p[T]/(Q)$, we combine linear algebra over $(R/(p))[T]/(Q)$ and Newton iteration to invert matrices in time $\mathcal{O}((n^\omega \mathsf{I}(N) + n^\Omega)\mathsf{M}(d))$, where $d := \deg_T(Q)$.

In this chapter, we denote by $\boldsymbol{f} = (f_1, ..., f_n) \in R[X_1, ..., X_n]$ a polynomial system given by an s.l.p. with L operations in $\{+, -, *\}$. If L_{f_i} is the evaluation complexity of only the output f_i, then we denote by $L^{\perp} := L_{f_1} + \cdots + L_{f_n}$ the complexity that corresponds to computing f_1, ..., f_n independently, that is without sharing any operations between the computation of different outputs f_i. Since $L_{f_i} \leqslant L$, we always have

$$L \leqslant L^{\perp} \leqslant n\, L.$$

When $\boldsymbol{f}$ is given as an s.l.p., its Jacobian matrix can be computed by an algorithm from Baur and Strassen [BS83]. This method uses $\mathcal{O}(L_{f_i})$ arithmetic operations to compute the gradient of f_i. Therefore, the Jacobian matrix of $\boldsymbol{f}$ can be evaluated in time $\mathcal{O}(L^{\perp})$.

6.1.2 Motivations

Lifting triangular sets (or univariate representations) is a crucial operation. Most implementations of algorithms that compute triangular set on rationals compute this object modulo a prime number, and then lift the representation. For example, the Kronecker software [L+02] for univariate representations and the RegularChains package [LMX05] of Maple for triangular sets use a lifting. Even better, the geometric resolution algorithm [GLS01, HMW01] which is implemented in Kronecker requires yet another lifting: a lifting on power series is employed to compute univariate representations of curves, which is a basic step of the algorithm.

As it turns out, most of the time required to compute triangular sets (or univariate representations) is spend in the lifting. Therefore, any improvement on the lifting complexity will have repercussions on the whole algorithm.

It was shown in Chapter 5 that relaxed algorithms could reduce the cost due to linear algebra when lifting a regular root of a polynomial system compared to offline, or zealous, algorithms. In the same way that the Newton iteration was adapted to lift univariate representations in [GLS01, HMW01] and then triangular sets in [Sch02], we adapt our relaxed approach to lift such objects with the hope of getting rid of the contribution of linear algebra in the complexity.

6.1.3 Results

Let $\boldsymbol{f} = (f_1, ..., f_n)$ be a polynomial system in $R[X_1, ..., X_n]$ and $\boldsymbol{t}_0$ be a triangular set in $R/(p)[X_1, ..., X_n]$ such that:

- $\boldsymbol{f}$ is given as an s.l.p. with inputs X_1, ..., X_n and n outputs corresponding to $f_1, ..., f_n$. This s.l.p. has operations in $\{+, -, *\}$ and can use constants in $A/\langle \boldsymbol{t}_0 \rangle$;

- $\boldsymbol{f} = 0$ in $R/(p)[X_1, ..., X_n]/\langle \boldsymbol{t}_0 \rangle$;
- the determinant of the Jacobian matrix $\mathrm{Jac}_{\boldsymbol{f}}$ in $\mathcal{M}_n(R/(p)[X_1, ..., X_n])$ must be invertible modulo $\boldsymbol{t}_0$.

This last condition is sufficient to have the existence and uniqueness of a triangular set $\boldsymbol{t}$ in $R_p[X_1, ..., X_n]$ which reduces to $\boldsymbol{t}_0$ modulo p and satisfies $\boldsymbol{f} = 0$ in $R_p[X_1, ..., X_n]/\langle \boldsymbol{t} \rangle$. From these inputs, we compute at some precision N this unique triangular set $\boldsymbol{t}$. We call this operation the lifting of the triangular set $\boldsymbol{t}$ at precision N.

Example 6.1. We consider the polynomial system $\boldsymbol{f} = (f_1, f_2)$ in $\mathbb{Z}[X_1, X_2]$ with

$$\begin{aligned} f_1 &:= 33\,X_2^3 + 14699\,X_2^2 + 276148\,X_1 + 6761112\,X_2 - 11842820 \\ f_2 &:= 66\,X_1\,X_2 + X_2^2 - 94\,X_1 - 75\,X_2 - 22. \end{aligned}$$

Let $\boldsymbol{t}_0$ be the triangular set of $(\mathbb{Z}/7\,\mathbb{Z})[X_1, X_2]$ given by

$$\boldsymbol{t}_0 := (X_1^2 + 5\,X_1, 3\,X_1\,X_2 + X_2^2 + 4\,X_1 + 2\,X_2 + 6).$$

We lift the triangular set $\boldsymbol{t}_0$ from $(\mathbb{Z}/7\,\mathbb{Z})[X_1, X_2]$ to a triangular set $\boldsymbol{t}$ in $\mathbb{Z}_7[X_1, X_2]$. At each step of the relaxed lifting, we increment the precision. So at the first step, we have

$$\boldsymbol{t} = (X_1^2 + (5 + 5 \cdot 7)\,X_1 + 7, (3 + 2 \cdot 7)\,X_1\,X_2 + X_2^2 + 4\,X_1 + (2 + 3 \cdot 7)\,X_2 + (6 + 3 \cdot 7))$$

in $(\mathbb{Z}_7/7^2\,\mathbb{Z}_7)[X_1, X_2]$. We iterate again and find

$$\begin{aligned} \boldsymbol{t} = (&X_1^2 + (5 + 5 \cdot 7 + 6 \cdot 7^2)\,X_1 + (7 + 7^2), \\ &(3 + 2 \cdot 7 + 7^2)\,X_1\,X_2 + X_2^2 + (4 + 5 \cdot 7^2)\,X_1 + (2 + 3 \cdot 7 + 5 \cdot 7^2)\,X_2 + \\ &(6 + 3 \cdot 7 + 6 \cdot 7^2)) \end{aligned}$$

in $(\mathbb{Z}_7/7^3\,\mathbb{Z}_7)[X_1, X_2]$. The precision is enough to recover the triangular set

$$\boldsymbol{t} := (X_1^2 - 9\,X_1 + 56, 66\,X_1\,X_2 + X_2^2 - 94\,X_1 - 75\,X_2 - 22) \in \mathbb{Z}[X_1, X_2].$$

Theorem 6.2. *With the former notations and hypotheses, we can lift the triangular set* $\boldsymbol{t}$ *at precision* N *in time*

$$[\mathcal{O}(n\,L\,\mathsf{R}(N)) + n^2 \log(n)^{\mathcal{O}(1)}\,N + \mathcal{O}(n^\Omega)]\,\mathsf{Rem}(d_1, ..., d_n).$$

A different technique improves the dominant asymptotic cost in the precision $n\,L\,\mathsf{R}(N)\,\mathsf{Rem}(d_1, ..., d_n)$ when the number e of essential variables is lower than n. This technique requires to solve a linear system where the matrix has finite precision. Since the definition of this matrix σ_B is quite technical, we just content ourselves with saying that its finite length, denoted by λ, satisfies $\lambda = \tilde{\mathcal{O}}(L\,d_1 \cdots d_n)$ and with

pointing to Formula 6.13 for a recursive definition of the rows of the matrix. In the special case where arithmetic operations in R_p have no carries, this length reduces to $\lambda = 1$.

Theorem 6.3. *We keep the former notations and hypotheses. We can lift the triangular set $\boldsymbol{t}$ at precision N in time*

$$\mathcal{O}([e\,L\,\mathsf{R}(N) + N\,\mathsf{MMR}(n,1,\lambda)/\lambda + n\,L\,N + n^{\Omega}]\,\mathsf{Rem}(d_1,...,d_n)),$$

that is

$$[\mathcal{O}(e\,L\,\mathsf{R}(N)) + n^2 \log(n)^{\mathcal{O}(1)} \log(\lambda)^{\mathcal{O}(1)}\,N + \mathcal{O}(n\,L\,N + n^{\Omega})]\,\mathsf{Rem}(d_1,...,d_n)$$

where λ satisfies $\lambda = \tilde{\mathcal{O}}(L\,d_1 \cdots d_n)$.

We deduce the following important corollary for univariate representations.

Corollary 6.4. (of Theorem 6.3) *Let $\boldsymbol{f} = (f_1, ..., f_n)$ be a polynomial system in $R[X_1, ..., X_n]$ given by an s.l.p. Γ and $\mathfrak{P}_0 = (Q_0, S_{1,0}, ..., S_{n,0})$ a univariate representation in $R/(p)[X_1, ..., X_n]$ such that $f(S_{1,0}, ..., S_{n,0}) = 0$ in $R/(p)[X]/Q_0$.*

Then there exists an integer λ satisfying $\lambda = \tilde{\mathcal{O}}(L\,d)$ such that we can lift the univariate representation $\mathfrak{P}$ at precision N in time

$$\mathcal{O}([L\,\mathsf{R}(N) + N\,\mathsf{MMR}(n,1,\lambda)/\lambda + n\,L\,N + n^{\Omega}]\,\mathsf{M}(d)).$$

Let us compare the relaxed approach to the off-line methods of Section 6.3. We focus on the asymptotic behavior in the precision N. For triangular sets, we have to compare the relaxed cost $n\,L\,\mathsf{R}(N)\,\mathsf{Rem}(d_1,...,d_n)$ to the zealous bound $\mathcal{O}(((L^{\perp} + n^{\omega})\,\mathsf{I}(N) + n^{\Omega})\,\mathsf{Rem}(d_1,...,d_n))$. In this case, we can hope for an improvement only when $n\,L \ll n^{\omega}$ and for precisions N where the ratio $\mathsf{R}(N)/\mathsf{I}(N)$ is moderate.

The relaxed approach for univariate representations is more profitable. The relaxed cost $L\,\mathsf{R}(N)\,\mathsf{M}(d_n)$ always compares favorably to the zealous cost $\mathcal{O}((L^{\perp} + n^{\Omega})\,\mathsf{I}(N)\,\mathsf{M}(d_n))$ for precisions N where the ratio $\mathsf{R}(N)/\mathsf{I}(N)$ is moderate.

6.2 Quotient and remainder modulo a triangular set

This section deals with Euclidean division modulo a triangular set. From now on, we denote by $\boldsymbol{t} = (t_1, ..., t_n)$ a triangular set of $R[X_1, ..., X_n]$. Computing remainders is a basic operation necessary to be able to compute with the quotient algebra $A := R[X_1, ..., X_n]/\langle \boldsymbol{t} \rangle$. We are also interested in the quotients of the division since we will need them later.

We start by defining quotients and remainder of the Euclidean division by $\boldsymbol{t}$ in a unique manner. Then we focus on computing this objects. We circumvent the fact that the size of the quotients is exponential in the size $d_1 \cdots d_n$ of the quotient algebra A by computing only reductions of the quotients modulo a triangular set. This leads us to Algorithms `Rem_triangular` and `Rem_quo_triangular`.

Canonical quotients and remainder For any $P \in R[X_1, ..., X_n]$, the existence of $r, q_1, ..., q_n \in R[X_1, ..., X_n]$ satisfying $P = r + q_1 t_1 + \cdots + q_n t_n$ and $\deg_{X_i}(r) < d_i$ is guaranteed because the elements of a triangular set are monic. The quotients $q_1, ...,$ q_n are not unique. For $1 \leqslant i < j \leqslant n$, let $z_{i,j}$ be the vector of $R[X_1, ..., X_n]^n$ with only t_j in the i-th position and $-t_i$ in the j-th position. We can add to any choice of quotients an element of the syzygy $R[X_1, ..., X_n]$-module spanned by the $(z_{i,j})_{1 \leqslant i < j \leqslant n}$ in $R[X_1, ..., X_n]^n$. Nevertheless, a canonical choice of quotient can be made, as for the division by a standard, or Gröbner, basis

Lemma 6.5. *For all $P \in R[X_1, ..., X_n]$, there exists a unique vector of polynomials $(r, q_1, ..., q_n)$ in $R[X_1, ..., X_n]^{n+1}$ such that*

$$P = r + q_1 t_1 + \cdots + q_n t_n$$

and for all $1 \leqslant i \leqslant n$, $\deg_{X_i}(r) < d_i$ and for all $1 \leqslant i < j \leqslant n$, $\deg_{X_j}(q_i) < d_j$.

Proof. Take any Euclidean decomposition $P = r + q_1 t_1 + \cdots + q_n t_n$ with $\deg_{X_i}(r) < d_i$. Then use the syzygies $(z_{1,i})_{1<i\leqslant n}$ to reduce the degree of q_1 in $X_2, ..., X_n$. Again use the syzygies $(z_{2,i})_{2<i\leqslant n}$ to reduce the degree of q_2 in $X_3, ..., X_n$. This last action do not change q_1. Continuing the process until we reduce the degree of q_{n-1} in X_n by $z_{n-1,n}$, we have exhibited a Euclidean decomposition satisfying the hypothesis of the lemma.

Now let us prove the uniqueness of $(r, q_1, ..., q_n)$. Because r is unique, we have to prove that if $q_1 t_1 + \cdots + q_n t_n = 0$ with $\deg_{X_j}(q_i) < d_j$ for all $1 \leqslant i < j \leqslant n$, then $q_1 = \cdots = q_n = 0$. By contradiction, we suppose there is such a decomposition with a non-zero q_i. Let j be the maximal index of a non-zero q_j. Then $\deg_{X_j}(q_j t_j) \geqslant d_j$ and in the same time

$$\begin{aligned}
\deg_{X_j}(q_j t_j) &= \deg_{X_j}(q_1 t_1 + \cdots + q_{j-1} t_{j-1}) \\
&\leqslant \max_{i<j}(\deg_{X_j}(q_i t_i)) \\
&\leqslant \max_{i<j}(\deg_{X_j}(q_i)) \\
&< d_j.
\end{aligned}$$

Contradiction. □

We call canonical quotients and remainder, those which satisfies the conditions of Lemma 6.5. We denote by $P \operatorname{rem} \boldsymbol{t}$ the remainder of P modulo $\boldsymbol{t}$. We can not compute the q_i because they suffer from the phenomenon of *intermediate expression swell*; in the computations of the remainder of a polynomial modulo a triangular set, the size of intermediate expressions, e.g. the size of the quotients, increases too

much. A quick estimate gives that the size of the quotients is exponential in the size $d_1 \cdots d_n$ of the quotient algebra A.

Fast multivariate Euclidean division by a triangular set Therefore we use a variant of the remainder algorithm of [LMMS09] that do not compute the entire quotients but modular reductions of them. Then we describe a second algorithm that keeps the quotient modulo another triangular set, avoiding once again to pay the cost due to their sizes.

We mention a different approach to compute remainders modulo a triangular set whose basic idea is to an evaluation / interpolation on the points of the variety defined by the triangular set [BCHS11]. The motivation behind this approach is to circumvent the exponential factor in the complexity. But because this approach can not be adapted to obtain the quotients, we will not use it here.

We denote by $d_i := \deg_{X_i}(t_i)$ the degree in X_i. For the sake of simplicity in our forthcoming algorithms, we will assume that the set of indices $\{i | d_i > 1\}$ of essential variables is $\{1, ..., e\}$, so that any reduced normal form r belongs to $R[X_1, ..., X_e]$.

Our algorithm `Rem_triangular` is a slight improvement of the algorithm of [LMMS09]: it does $3\,d_e$ recursive calls instead of $4\,d_e$. As a consequence, the exponential factor in the complexity is 3^e instead of 4^e. Algorithm `Rem_triangular` is meant to reduce the product of two reduced elements. Therefore we suppose that the input polynomial $P \in R[X_1, ..., X_e]$ satisfies $\deg_{X_i}(P) \leqslant 2\,(d_i - 1)$.

The forthcoming algorithm is a triangular version of the fast univariate division with remainder (see [GG03, Section 9.1]). If $P \in R[X_1, ..., X_e]$, we denote by $P[X_e^i] \in R[X_1, ..., X_{e-1}]$ the coefficient of P in X_e^i. If $\deg_{X_e}(P) = d$, we denote by $\mathrm{rev}_{X_e}(P)$ its reverse polynomial w.r.t. X_e defined by $\mathrm{rev}_{X_e}(P) := \sum_{i=0}^{d} (P[X_e^{d-i}])\, X_e^i$.

Algorithm `Rem_triangular`

Input: $\boldsymbol{t}$ and $P \in R[X_1, ..., X_e]$ such that $\deg_{X_i}(P) \leqslant 2\,(d_i - 1)$
Output: $r \in R[X_1, ..., X_e]$ reduced modulo $\boldsymbol{t}$ such that

$$r = P \operatorname{rem} \boldsymbol{t}.$$

1. Let $\boldsymbol{t}' := (t_1, ..., t_{e-1})$ and $R' := R[X_1, ..., X_{e-1}]/\langle \boldsymbol{t}' \rangle$.
   Compute the quotient q_e of P by t_e in $R'[X_e]$:
   a. $q_e := \sum_{i=d_e}^{2d_e-1}$ `Rem_triangular`$(\boldsymbol{t}', P[X_e^i])\, X_e^i$
   b. Precompute $I := 1/\mathrm{rev}_{X_e}(t_e) \operatorname{rem} X_e^{d_e-1}$ in $R'[X_e]$
   c. $q_e := \mathrm{rev}_{X_e}(q_e)\, I \operatorname{rem} X_e^{d_e}$ in $R'[X_e]$
   d. $q_e := \mathrm{rev}_{X_e}(q_e)$
2. $r := (P - q_e\, t_e) \operatorname{rem} X_e^{d_e}$ in $R[X_1, ..., X_{e-1}][X_e]$
3. $r := \sum_{i=0}^{d_e-1}$ `Rem_triangular`$(r[X_e^i], \boldsymbol{t}')\, X_e^i$
4. **return** r

The precomputation of step 1.b means that, as the object I depends only on $\boldsymbol{t}$, we compute it once and for all at the first call of Algorithm `Rem_triangular`.

In accord with the introduction, we denote by $\mathsf{Rem}(d_1, ..., d_e)$ the complexity of Algorithm `Rem_triangular`.

Proposition 6.6. *The algorithm* `Rem_triangular` *is correct and runs in time* $\mathsf{Rem}(d_1, ..., d_e) = \mathcal{O}(\mathsf{M}(3^e\, d_1 \cdots d_e))$.

Proof. Since Algorithm `Rem_triangular` is very similar to their algorithm, we refer to [LMMS09] for the proof of correction of our algorithm.

Step 1.c involves a multiplication in $R[X_1, ..., X_e]$ and a reduction by $\boldsymbol{t}'$ of the coefficients in X_e^i for $i < d_e$. So multivariate multiplications are used in steps 1.c and 2 and d_e reductions by $\boldsymbol{t}'$ are done in steps 1.a, 1.c and 2. Thus the complexity analysis becomes

$$\mathsf{Rem}(d_1, ..., d_e) \;=\; 3\, d_e\, \mathsf{Rem}(d_1, ..., d_{e-1}) + 2\, \mathsf{M}(d_1, ..., d_e),$$

and

$$\begin{aligned}
& \mathsf{Rem}(d_1, ..., d_e) \;=\; \mathcal{O}\left(\sum_{i=1}^{e} 3^{e-i}\, \mathsf{M}(d_1, ..., d_i)\, d_{i+1} \cdots d_e \right) \\
\Rightarrow\quad & \mathsf{Rem}(d_1, ..., d_e) \;=\; \mathcal{O}\left(\sum_{i=1}^{e} 3^{e-i}\, \mathsf{M}(2^i\, d_1 \cdots d_i)\, d_{i+1} \cdots d_e \right) \\
\Rightarrow\quad & \mathsf{Rem}(d_1, ..., d_e) \;=\; \mathcal{O}\left(\sum_{i=1}^{e} \mathsf{M}\left(3^e \left(\frac{2}{3} \right)^i d_1 \cdots d_e \right) \right) \\
\Rightarrow\quad & \mathsf{Rem}(d_1, ..., d_e) \;=\; \mathcal{O}(\mathsf{M}(3^e\, d_1 \cdots d_e)).
\end{aligned}$$

The precomputation of step 1.b costs $\mathcal{O}(\mathsf{M}(d_1, ..., d_e) + d_e\ \mathsf{Rem}(d_1, ..., d_{e-1}))$ by Newton iteration for the inversion, that is $\mathcal{O}(\mathsf{Rem}(d_1, ..., d_e))$. □

Remark that non-essential variables do not impact the complexity of our algorithm, *i.e.* $\mathsf{Rem}(d_1, ..., d_e, 1, ..., 1) = \mathsf{Rem}(d_1, ..., d_e)$. Indeed if $d_i = 1$, then we condition $\deg_{X_i}(P) \leqslant 2\,(d_i - 1)$ implies that P does not depend on X_i.

Recall that since the product of two reduced element in the quotient algebra $R[X_1, ..., X_e]/\langle \boldsymbol{t} \rangle$ satisfies the degree condition of the input of Algorithm `Rem_triangular`, arithmetic operations in this quotient algebra cost $\mathsf{M}(d_1, ..., d_n) + \mathsf{Rem}(d_1, ..., d_n)$, that is $\mathcal{O}(\mathsf{Rem}(d_1, ..., d_n))$.

Now we adapt Algorithm `Rem_triangular` to keep the quotients modulo another triangular set $\boldsymbol{t}^2$. In order to simplify the algorithm and because it suits our future needs, we assume that for all i, $\deg_{X_i}(t_i^1) = \deg_{X_i}(t_i^2)$ and still denote by d_i this degree.

Algorithm `Rem_quo_triangular`

Input:

- Triangular sets $\boldsymbol{t}^1$, $\boldsymbol{t}^2$
- $P \in R[X_1, ..., X_e]$ such that $\deg_{X_i}(P) \leqslant 2(d_i - 1)$

Output: $r, q_1, ..., q_e \in R[X_1, ..., X_e]$ reduced modulo $\boldsymbol{t}^1$ (or $\boldsymbol{t}^2$) such that

$$P = r + \sum_{i=1}^{e} q_i\, t_i^1 \quad \text{modulo } \langle \boldsymbol{t}^1 \rangle \langle \boldsymbol{t}^2 \rangle.$$

1. Let $\boldsymbol{t}^{2'} := (t_1^2, ..., t_{e-1}^2)$ and $R' := R[X_1, ..., X_{e-1}]/\langle \boldsymbol{t}^{2'} \rangle$. Compute the quotient q_e of P by t_e^1 in $R'[X_e]$:
   a. $q_e := \sum_{i=d_e}^{2d_e-1}$ `Rem_triangular`$(\boldsymbol{t}^{2'}, P[X_e^i])\, X_e^i$
   b. Precompute $I := 1/\mathrm{rev}_{X_e}(t_e^1) \operatorname{rem} X_e^{d_e-1}$ in $R'[X_e]$
   c. $q_e := (\mathrm{rev}_{X_e}(q_e)\, I) \operatorname{rem} X_e^{d_e}$ in $R'[X_e]$
   d. $q_e := \mathrm{rev}_{X_e}(q_e)$
2. $r := (P - q_e\, t_e^1)$ in $R[X_1, ..., X_e]$
3. $r_1 := r \operatorname{rem} X_e^{d_e}$ and $r_2 = r - r_1$
4. $0, q_1, ..., q_{e-1} := \sum_{i=d_e}^{2d_e-1}$ `Rem_quo_triangular`$(r_2[X_e^i], \boldsymbol{t}^{2'}, (t_1^1, ..., t_{e-1}^1))\, X_e^i$
5. **for** i from 1 to $e-1$
   $q_i' :=$ `Rem_triangular`$(q_i, \boldsymbol{t}^1)$
6. $r := r_1 + q_1' t_1^2 + \cdots + q_{e-1}' t_{e-1}^2$ in $R[X_1, ..., X_e]$
7. $r, q_1, ..., q_{e-1} := \sum_{i=0}^{d_e-1}$ `Rem_quo_triangular`$(r[X_e^i], \boldsymbol{t}^{1'}, \boldsymbol{t}^{2'})\, X_e^i$
8. **return** $r, q_1, ..., q_{e-1}, q_e$

We denote by $\mathsf{RemQuo}(d_1, ..., d_e)$ the complexity of `Rem_quo_triangular` for triangular sets $\boldsymbol{t}^1$ and $\boldsymbol{t}^2$ of same degrees $d_1, ..., d_e$.

Lemma 6.7. *If r is reduced modulo $\boldsymbol{t}^1$ and $P = r + \sum_{i=1}^{e} q_i\, t_i^1$ modulo the product ideal $\langle \boldsymbol{t}^1 \rangle \langle \boldsymbol{t}^2 \rangle$, then r is the reduced normal form of P modulo $\boldsymbol{t}^1$ and*

$$P = r + \sum_{i=1}^{e} q_i\, t_i^1 \quad \text{modulo } \boldsymbol{t}^2.$$

Proof. Since the product ideal $\langle \boldsymbol{t}^1 \rangle \langle \boldsymbol{t}^2 \rangle$ is included in both the ideals $\langle \boldsymbol{t}^1 \rangle$ and $\langle \boldsymbol{t}^2 \rangle$, the relation $P = r + \sum_{i=1}^{e} q_i\, t_i^1$ stands modulo both these ideals. So $P = r$ modulo $\boldsymbol{t}^1$ and since r is reduced, it is the reduced normal form of P. □

Proposition 6.8. *The algorithm* `Rem_quo_triangular` *is correct and its costs verifies* $\mathsf{RemQuo}(d_1, ..., d_e) = \mathcal{O}(e\, \mathsf{Rem}(d_1, ..., d_e))$.

Proof. We proceed recursively on the number e on variables involved in P. In one variable, our algorithm coincides with the fast univariate division with remainder (see [GG03, Section 9.1]). So it is correct and $\mathsf{RemQuo}(d_1) = \mathsf{Rem}(d_1)$.

Let's suppose that we have proved our claims in less that e variables. Since q_e is the quotient of P by t_e^1 in $(R[X_1, ..., X_{e-1}]/\langle \boldsymbol{t}^{2'} \rangle)[X_e]$, we have $r_2 = 0$ in $(R[X_1, ..., X_{e-1}]/\langle \boldsymbol{t}^{2'} \rangle)[X_e]$. By assumption, the recursive call of step 4 gives the decomposition

$$r_2 = \sum_{i=1}^{e-1} q_i\, t_i^2 \quad \text{modulo } \langle \boldsymbol{t}^{1'} \rangle\, \langle \boldsymbol{t}^{2'} \rangle$$

and also modulo $\langle \boldsymbol{t}^1 \rangle\, \langle \boldsymbol{t}^2 \rangle$. The reduction of the quotient of step 5 gives

$$r_2 = \sum_{i=1}^{e-1} q_i'\, t_i^2 \quad \text{modulo } \langle \boldsymbol{t}^1 \rangle\, \langle \boldsymbol{t}^2 \rangle$$

where the polynomials q_i' are reduced modulo $\boldsymbol{t}^1$. Therefore the polynomial $r_2' := \sum_{i=1}^{e-1} q_i'\, t_i^2$ has degree $\deg_{X_e}(r_2') < d_e$ and $\deg_{X_i}(r_2') < 2\, d_i$ for $i < e$. Because r_1 satisfies the same degree conditions, they are still satisfied by $r = r_1 + r_2'$. By the induction hypothesis, at step 7, we have for all $0 \leqslant i < d_e$ and $1 \leqslant j \leqslant e-1$, that $q_j[X_e^i]$ is reduced modulo $\boldsymbol{t}^{1'}$. Since q_j has degree less than d_e in X_e, it is reduced modulo $\boldsymbol{t}^1$. The last quotient q_e is also reduced because it was computed in $(R[X_1, ..., X_{e-1}]/\langle \boldsymbol{t}^{2'} \rangle)[X_e]$ and $\deg_{X_e}(q_e) = \deg_{X_e}(P) - \deg_{X_e}(t_e^1) < d_e$. Finally

$$\begin{aligned} P = r_1 + r_2 + q_e\, t_e^1 &= (r_1 + r_2') + q_e\, t_e^1 && \text{modulo } \langle \boldsymbol{t}^1 \rangle\, \langle \boldsymbol{t}^2 \rangle \\ &= \left(r + \sum_{i=1}^{e-1} q_i\, t_i^1 \right) + q_e\, t_e^1 && \text{modulo } \langle \boldsymbol{t}^1 \rangle\, \langle \boldsymbol{t}^2 \rangle. \end{aligned}$$

Concerning the complexity analysis, we have

$$\begin{aligned} \mathsf{RemQuo}(d_1, ..., d_e) &= 2\, d_e\, \mathsf{RemQuo}(d_1, ..., d_{e-1}) + 2\, d_e\, \mathsf{Rem}(d_1, ..., d_{e-1}) + \\ &\quad (e-1)\, \mathsf{Rem}(d_1, ..., d_e) + (e+1)\, \mathsf{M}(d_1, ..., d_e) \end{aligned}$$

which gives

$$\mathsf{RemQuo}(d_1, ..., d_e) = \mathcal{O}(e\, \mathsf{Rem}(d_1, ..., d_e)). \qquad \square$$

As for the remainder algorithm, we have $\mathsf{RemQuo}(d_1, ..., d_e, 1, ..., 1)$ equals to $\mathsf{RemQuo}(d_1, ..., d_e)$, so that

$$\mathsf{RemQuo}(d_1, ..., d_n) = \mathsf{RemQuo}(d_1, ..., d_e) = \mathcal{O}(e\, \mathsf{Rem}(d_1, ..., d_e)) = \mathcal{O}(e\, \mathsf{Rem}(d_1, ..., d_n)).$$

Also we notice that the remainder and quotients modulo $\langle \boldsymbol{t}^1 \rangle\, \langle \boldsymbol{t}^2 \rangle$ of the product of two reduced element in A costs $\mathcal{O}(\mathsf{RemQuo}(d_1, ..., d_e))$.

When we apply Algorithm `Rem_quo_triangular` to the triangular sets $\boldsymbol{t}$ and $\boldsymbol{t}_0$ in Section 6.4, the reductions modulo $\boldsymbol{t}^2 = \boldsymbol{t}_0$ will be cheaper than reductions modulo $\boldsymbol{t}^1 = \boldsymbol{t}$ since they can be done coefficientwise. However the overall costs of Algorithm `Rem_quo_triangular`$(\boldsymbol{t}, \boldsymbol{t}_0, \_)$ will remain bounded by $\mathcal{O}(e)$ times the cost of reduction by $\boldsymbol{t}$.

Finally, we point out the situation would have been different with naïve algorithms for the remainder and quotients. The naïve remainder algorithm reduces the leading terms in X_e one by one, as would do a Gröbner basis reduction algorithm for the lexicographical monomial ordering with $X_1 \ll \cdots \ll X_n$. This algorithm implicitly computes the whole quotients and therefore $\mathsf{RemQuo}(d_1, \ldots, d_n) = \mathsf{Rem}(d_1, \ldots, d_n)$ with naïve algorithms.

Shift index The shift index is a theoretical tool used to prove the correctness of the computation of recursive p-adic numbers (see Proposition 2.17). We assess the shift index of the two previous algorithms with respect to the p-adic coefficients of the triangular sets.

Lemma 6.9. *Let Γ be an s.l.p. with n inputs and one output which satisfies $\mathrm{sh}(\Gamma) \geqslant 0$. Let $\Gamma(\boldsymbol{t})$ denote the output of Γ on the inputs $\boldsymbol{t}$.*

Then one has, for any triangular set $\boldsymbol{t}^2$,

$$\mathrm{sh}(\boldsymbol{t} \mapsto \texttt{Rem\_triangular}\,(\boldsymbol{t}, \Gamma(\boldsymbol{t}))) \geqslant 0$$

$$\mathrm{sh}(\boldsymbol{t} \mapsto \texttt{Rem\_quo\_triangular}\,(\boldsymbol{t}, \boldsymbol{t}^2, \Gamma(\boldsymbol{t}))) \geqslant 0.$$

In other words, Lemma 6.9 states that if the nth p-adic coefficient of a polynomial $\Gamma(\boldsymbol{t})$ involves only the ith coefficients of $\boldsymbol{t}$ for $i \leqslant n$, then so it is for the polynomials `Rem_triangular` $(\boldsymbol{t}, \Gamma(\boldsymbol{t}))$ and `Rem_quo_triangular` $(\boldsymbol{t}, \boldsymbol{t}^2, \Gamma(\boldsymbol{t}))$. The notation $\boldsymbol{t} \mapsto \texttt{Rem\_triangular}\,(\boldsymbol{t}, \Gamma(\boldsymbol{t}))$ refers to an s.l.p. which takes as input $\boldsymbol{t}$ and outputs `Rem_triangular` $(\boldsymbol{t}, \Gamma(\boldsymbol{t}))$ (see Remark 2.2). The entries $\boldsymbol{t}$ are given by the list of their polynomial coefficients, so that we can reverse polynomials.

Proof. We prove it for `Rem_quo_triangular`, the other case being similar. We proceed by induction on the number n of variables involved in the input of $\boldsymbol{t}^1$. If no variables are involved, then our algorithm does nothing and its shift index is the one of Γ. From now on, let us assume that the result is valid for input of less than n variables.

First, we prove that the computations that leads to $I := 1/\mathrm{rev}_{X_n}(t_n) \operatorname{rem} X_n^{d_n - 1}$ in $R'[X_n]$ have a non-negative shift. Define $I_0 := 1$ and $I_\ell := I_{\ell-1} - I_{\ell-1}\,(\mathrm{rev}_{X_\ell}(t_\ell)\, I_{\ell-1} - 1)$ in $R'[X_n]/\langle X_n^\ell \rangle$. Thereby $I = I_{\lceil \log_2(d_n - 1) \rceil}$ modulo $X_n^{d_n - 1}$. Since I_0 has a non-negative shift index and since I_ℓ is obtained from $I_{\ell-1}$ by multiplication and reduction modulo $\boldsymbol{t}'$ of operands which have a non-negative shift, we deduce that I has itself a non-negative shift by the induction hypothesis.

Our algorithm uses only recursive calls in less variables, addition and multiplication. Since all these operations preserve a non-negative shift, we deduce that q_e, r and finally $r, q_1, \ldots, q_e$ have non-negative shift indices. □

6.3 Overview of off-line lifting algorithms

We present three existing lifting algorithms, which are off-line, in increasing order of generality (and complexity). First algorithm lifts only a regular root, so it applies only to triangular sets with $d_1 = \cdots = d_n = 1$. Second algorithm lifts a univariate representation, that is a triangular set with $d_1 = \cdots = d_{n-1} = 1$ and any degree d_n. And finally we present an algorithm that lift any triangular set.

6.3.1 Hensel-Newton local lifting of a root

We start by recalling the local Newton iterator, that lift a regular root of an algebraic system into the completion ring R_p. It was first introduced by [New36] for finding power series solutions of univariate polynomials with coefficients in $\Bbbk[[X]]$. This method allows a local study of an algebraic variety. A relaxed version of this algorithm is presented in Chapter 5.

We detail the Newton iteration that doubles the precision of a regular solution of the algebraic system $\boldsymbol{f}$.

Algorithm `Local_Newton_step`

Input:

- System of equation $\boldsymbol{f}$ and its Jacobian $\mathrm{Jac}_{\boldsymbol{f}}$ as an s.l.p.
- A root $\boldsymbol{S} = (S_1, ..., S_n)$ of $\boldsymbol{f}$ in $R/\left(p^{2^{m-1}}\right)$
- Inverse $\mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{S})$ of $\mathrm{Jac}_{\boldsymbol{f}}(\boldsymbol{S})$ in $\mathcal{M}_n\left(R/\left(p^{2^{m-2}}\right)\right)$

Output:

- A root $\boldsymbol{S}' = (S'_1, ..., S'_n)$ of $\boldsymbol{f}$ in $R/(p^{2^m})$
- Inverse $\mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{S})'$ of $\mathrm{Jac}_{\boldsymbol{f}}(\boldsymbol{S})$ in $\mathcal{M}_n\left(R/\left(p^{2^{m-1}}\right)\right)$

1. In $\mathcal{M}_n\left(R/\left(p^{2^{m-1}}\right)\right)$, compute

   $\mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{S})' := \mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{S}) - \mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{S})\left(\mathrm{Jac}_{\boldsymbol{f}}(\boldsymbol{S}) \cdot \mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{S}) - \mathrm{Id}_n\right)$
2. $\boldsymbol{S}' := \boldsymbol{S} - \mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{S})' \cdot \boldsymbol{f}(\boldsymbol{S})$ in $(R/(p^{2^m})[T])^n$
3. **return** $\boldsymbol{S}', \mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{S})'$

Proposition 6.10. *The algorithm* `Local_Newton_step_univariate` *is correct and costs* $\mathcal{O}((L^{\perp} + n^{\omega})\,\mathsf{I}(N) + n^{\Omega})$ *to lift a regular root in* R_p *at precision* N.

Proof. The proof of correctness is classical and can be found in [GG03]. The cost $\mathcal{O}(n^{\Omega})$ is to compute the inverse of the Jacobian matrix modulo p. The Jacobian matrix $\mathrm{Jac}_{\boldsymbol{f}}$ can be evaluated in $\mathcal{O}(L^{\perp})$ operations in $R/(p^{2^m})$, which each costs $\mathsf{I}(2^m)$, so the result follows. □

6.3.2 Hensel-Newton global lifting of univariate representation

The following algorithm lifts univariate representations under the condition that the Jacobian matrix is invertible modulo the univariate representation over $R/(p)$. This algorithm is a slight modification from the local Newton iterator. It was first introduced in [GLS01, HMW01] and generalizes the previous approach.

Algorithm `Global_Newton_step_univariate`

Input:

- System of equation $\boldsymbol{f}$ and its Jacobian $\mathrm{Jac}_{\boldsymbol{f}}$ as an s.l.p.
- $u = \lambda_1 X_1 + \cdots + \lambda_n X_n$ a linear form with $\lambda_i \in R$
- Univariate representation $\boldsymbol{S} = (S_1, \ldots, S_n)$ and Q in $R/\big(p^{2^{m-1}}\big)[T]$
- Inverse $\mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{S})$ of $\mathrm{Jac}_{\boldsymbol{f}}(\boldsymbol{S})$ in $\mathcal{M}_n\big(R/\big(p^{2^{m-2}}\big)[T]/Q\big)$

Output:

- Univariate representation $\boldsymbol{S}' = (S'_1, \ldots, S'_n)$ and Q' in $R/(p^{2^m})[T]$
- Inverse $\mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{S})'$ of $\mathrm{Jac}_{\boldsymbol{f}}(\boldsymbol{S})$ in $\mathcal{M}_n\big(R/\big(p^{2^{m-1}}\big)[T]/Q\big)$

1. In $\mathcal{M}_n\big(R/\big(p^{2^{m-1}}\big)[T]/Q\big)$, compute
$$\mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{S})' := \mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{S}) - \mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{S})\,(\mathrm{Jac}_{\boldsymbol{f}}(\boldsymbol{S}) \cdot \mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{S}) - \mathrm{Id}_n)$$
2. $\boldsymbol{S}' := (\boldsymbol{S} - \mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{S})' \cdot \boldsymbol{f}(\boldsymbol{S})) \operatorname{rem} Q$ in $(R/(p^{2^m})[T])^n$
3. $\Delta := u(\boldsymbol{S}') - T$ in $(R/(p^{2^m})[T])^n$
4. $\boldsymbol{S}' := \boldsymbol{S}' - \Big(\Big(\frac{\partial \boldsymbol{S}'}{\partial T}\Delta\Big) \operatorname{rem} Q\Big)$ in $(R/(p^{2^m})[T])^n$
5. $Q' := Q - \big(\big(\frac{\partial Q}{\partial T}\Delta\big) \operatorname{rem} Q\big)$ in $R/(p^{2^m})[T]$
6. **return** $\boldsymbol{S}'$, Q' and $\mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{S})'$

Proposition 6.11. *The algorithm* `Global_Newton_step_univariate` *is correct and costs* $\mathcal{O}((L^{\perp}+n^{\omega})\,\mathsf{M}(d)\,\mathsf{I}(N)+n^{\Omega})$ *to lift a univariate representation at precision* N *under the condition that* $\mathrm{Jac}_{\boldsymbol{f}}(\boldsymbol{S})$ *is invertible in* $R/(p)[T]/Q$.

We refer to [GLS01] for a proof of this proposition.

6.3.3 Hensel-Newton global lifting of triangular sets

The following algorithm was introduced in [Sch02]. Roughly speaking, it can be seen as a classical Newton iteration for finding a zero of the function $\Phi\colon \boldsymbol{Y} \mapsto B \cdot \boldsymbol{Y} - \boldsymbol{f}$ where B is an element of $\mathcal{M}_n(R[X_1, \ldots, X_n])$ satisfying $\boldsymbol{f} = B \cdot \boldsymbol{t}$. This algorithm lifts any triangular set under an inversibility condition of the Jacobian matrix. In the special case of univariate representations, this algorithm does the same computations as Algorithm `Global_Newton_step_univariate`, but they are presented differently, with only matrix multiplications, and more concisely.

Example 6.12. We consider the polynomial system $\boldsymbol{f} = (f_1, f_2)$ in $\mathbb{Z}[X_1, X_2]$ with

$$\begin{aligned} f_1 &:= 33\,X_2^3 + 14699\,X_2^2 + 276148\,X_1 + 6761112\,X_2 - 11842820 \\ f_2 &:= 66\,X_1\,X_2 + X_2^2 - 94\,X_1 - 75\,X_2 - 22. \end{aligned}$$

Let $\boldsymbol{t}_0$ be the triangular set of $(\mathbb{Z}/7\,\mathbb{Z})[X_1, X_2]$ given by

$$\boldsymbol{t}_0 := (X_1^2 + 5\,X_1, 3\,X_1\,X_2 + X_2^2 + 4\,X_1 + 2\,X_2 + 6).$$

We lift the triangular set $\boldsymbol{t}_0$ from $(\mathbb{Z}/7\,\mathbb{Z})[X_1, X_2]$ to a triangular set $\boldsymbol{t}$ in $\mathbb{Z}_7[X_1, X_2]$. At each step of the off-line lifting, we double the precision. So at the first step, we have

$$\boldsymbol{t} = (X_1^2 + 40\,X_1 + 7, 17\,X_1\,X_2 + X_2^2 + 4\,X_1 + 23\,X_2 + 27) \in (\mathbb{Z}/7^2\,\mathbb{Z})[X_1, X_2].$$

We iterate again and find

$$\boldsymbol{t} = (X_1^2 + 2392\,X_1 + 56, 66\,X_1\,X_2 + X_2^2 + 2307\,X_1 + 2326\,X_2 + 2379)$$

in $(\mathbb{Z}/7^4\,\mathbb{Z})[X_1, X_2]$. The precision is enough to recover the triangular set

$$\boldsymbol{t} := (X_1^2 - 9\,X_1 + 56, 66\,X_1\,X_2 + X_2^2 - 94\,X_1 - 75\,X_2 - 22) \in \mathbb{Z}[X_1, X_2].$$

Algorithm `Global_Newton_step_triangular`

Input:

- System of equation $\boldsymbol{f}$ and its Jacobian $\mathrm{Jac}_{\boldsymbol{f}}$ as an s.l.p.
- Triangular set $\boldsymbol{t} = (t_1, ..., t_n)$ in $R/\big(p^{2^{m-1}}\big)[X_1, ..., X_n]$
- Inverse $\mathrm{IJac}_{\boldsymbol{t}}$ of $\mathrm{Jac}_{\boldsymbol{t}}$ in $\mathcal{M}_n\big(R/\big(p^{2^{m-2}}\big)[X_1, ..., X_n]/\langle \boldsymbol{t} \rangle\big)$

Output:

- Triangular set $\boldsymbol{t}' = (t'_1, ..., t'_n)$ in $R/(p^{2^m})[X_1, ..., X_n]$
- Inverse $\mathrm{IJac}_{\boldsymbol{f}}$ of $\mathrm{Jac}_{\boldsymbol{f}}$ in $\mathcal{M}_n\big(R/\big(p^{2^{m-1}}\big)[X_1, ..., X_n]/\langle \boldsymbol{t} \rangle\big)$

1. In $\mathcal{M}_n\big(R/\big(p^{2^{m-1}}\big)[X_1, ..., X_n]/\langle \boldsymbol{t} \rangle\big)$, compute
$$\mathrm{IJac}'_{\boldsymbol{f}} := \mathrm{IJac}_{\boldsymbol{f}} - \mathrm{IJac}_{\boldsymbol{f}}\,(\mathrm{Jac}_{\boldsymbol{f}} \cdot \mathrm{IJac}_{\boldsymbol{f}} - \mathrm{Id}_n)$$
2. $\delta\boldsymbol{t} := \mathrm{Jac}_{\boldsymbol{t}} \cdot \mathrm{IJac}_{\boldsymbol{f}}(\boldsymbol{t})' \cdot \boldsymbol{f}$ in $(R/(p^{2^m})[X_1, ..., X_n]/\langle \boldsymbol{t} \rangle)^n$
3. $\boldsymbol{t}' := \boldsymbol{t} + \delta\boldsymbol{t}$ in $(R/(p^{2^m})[X_1, ..., X_n]/\langle \boldsymbol{t} \rangle)^n$
4. **return** $\boldsymbol{t}'$

Proposition 6.13. *The algorithm* `Global_Newton_step_triangular` *is correct and costs* $\mathcal{O}((L^{\perp} + n^{\omega})\,\mathsf{Rem}(d_1, ..., d_n)\,\mathsf{I}(N))$ *to lift a triangular set at precision* N *under the condition that* $\mathrm{Jac}_{\boldsymbol{f}}$ *is invertible in* $R/(p)[X_1, ..., X_n]/\langle \boldsymbol{t} \rangle$.

6.4 Relaxed lifting of triangular sets

Let $\boldsymbol{t}_0$ be a triangular set of $R/(p)[X_1, ..., X_n]$. Define the $R/(p)$-algebra A_0 by $A_0 := R/(p)[X_1, ..., X_n]/\langle \boldsymbol{t}_0 \rangle$. Let $\boldsymbol{f}$ be given as an s.l.p. Γ with inputs $X_1, ..., X_n$ and n outputs corresponding to $f_1, ..., f_n$. The s.l.p. Γ has operations in $\{+, -, *\}$ and can use constants in $A/\langle \boldsymbol{t}_0 \rangle$. We assume that the triangular set $\boldsymbol{t}_0$ satisfies the property that $\mathrm{Jac}(\boldsymbol{f}_0)$ is invertible in $\mathcal{M}_n(A_0)$. Then there exists a unique triangular set $\boldsymbol{t}$ in $R_p[X_1, ..., X_n]$ which reduces to $\boldsymbol{t}_0$ modulo p and satisfies $\boldsymbol{f} = 0$ in $R_p[X_1, ..., X_n]/\langle \boldsymbol{t} \rangle$.

In this section we detail two relaxed algorithms that lift $\boldsymbol{t}$ at precision N. The first algorithm of Section 6.4.1 should be used for generic triangular set. We refine this algorithm in Section 6.4.2 for triangular sets with few essential variables, e.g. for univariate representations.

Throughout this section, we denote by $P_{-n}, ..., P_L$ the result sequence of the s.l.p. Γ on the input $X_1, ..., X_n$. Let r_i and $\boldsymbol{b}_i$ be the canonical remainder and quotients of P_i for $-n \leqslant i \leqslant L$. So we have $P_i = r_i + \boldsymbol{b}_i \, \boldsymbol{t} \in R[X_1, ..., X_n]$. Let $i_1, ..., i_n$ be the indices of the n outputs of Γ, so that we have $f_j = P_{i_j}$ for $1 \leqslant j \leqslant n$. We denote by $B \in \mathcal{M}_n(R[X_1, ..., X_n])$ the matrix whose jth row is $\boldsymbol{b}_{i_j}$. Therefore one has $\boldsymbol{f} = B \, \boldsymbol{t} \in \mathcal{M}_{n,1}(R[X_1, ..., X_n])$.

6.4.1 Using the quotient matrix

We define two maps σ and δ from R_p to R_p by $\sigma(a) = a_0$ and $\delta(a) := \frac{a - a_0}{p}$ for any $a = \sum_{i \in \mathbb{N}} a_i \, p^i \in R_p$. For any $a \in R_p$, we have $a = \sigma(a) + p \, \delta(a)$. We extend the definition of σ and δ to $A := R_p[X_1, ..., X_n]$ by mapping X_i to itself and to $\mathcal{M}_{r,s}(A)$ by acting componentwise. Thus $\sigma(\boldsymbol{t})$, also denoted by $\boldsymbol{t}_0$, is defined by $\sigma(\boldsymbol{t}) := (\sigma(t_1), ..., \sigma(t_n))$.

Recursive formula for $\boldsymbol{t}$ The triangular set $\boldsymbol{t}$ is a recursive p-adic vector of polynomials.

Lemma 6.14. *The matrix $\sigma(B) \in \mathcal{M}_n(R_p[X_1, ..., X_n])$ is invertible modulo $\boldsymbol{t}_0$. Moreover the triangular set $\boldsymbol{t}$ satisfies the recursive equation*

$$\boldsymbol{t} - \boldsymbol{t}_0 = \sigma(B)^{-1} \, (\boldsymbol{f} - p^2 \, (\delta(B) \cdot \delta(\boldsymbol{t}))) \text{ rem } \boldsymbol{t}_0 \tag{6.1}$$

in $\mathcal{M}_{n,1}(R_p[X_1, ..., X_n])$.

Proof. For any $P \in R[X_1, ..., X_n]$, let r and $\boldsymbol{a}$ be the canonical remainder and quotients of P by $\boldsymbol{t}$ so that

$$P - r = \boldsymbol{a} \cdot \boldsymbol{t} = \sigma(\boldsymbol{a}) \cdot \boldsymbol{t} + p \, \delta(\boldsymbol{a}) \cdot \boldsymbol{t} = \sigma(\boldsymbol{a}) \cdot \boldsymbol{t} + p \, \delta(\boldsymbol{a}) \cdot \boldsymbol{t}_0 + p^2 \, \delta(\boldsymbol{a}) \cdot \delta(\boldsymbol{t}).$$

Thus we have

$$\sigma(\boldsymbol{a}) \cdot \boldsymbol{t} = P - (r + p^2 \, \delta(\boldsymbol{a}) \cdot \delta(\boldsymbol{t}))$$

in $R[X_1, ..., X_n]/\langle \boldsymbol{t}_0 \rangle$. Now if $P \in \langle \boldsymbol{t} \rangle$, then $r = 0$ and we get

$$\sigma(\boldsymbol{a}) \cdot \boldsymbol{t} = P - p^2 \, (\delta(\boldsymbol{a}) \cdot \delta(\boldsymbol{t}))$$

in $R[X_1,...,X_n]/\langle \boldsymbol{t}_0\rangle$. We apply this to the equations $\boldsymbol{f}$ and get

$$\sigma(B)\cdot \boldsymbol{t} = \boldsymbol{f} - p^2\,(\delta(B)\cdot\delta(\boldsymbol{t})) \tag{6.2}$$

in $R[X_1,...,X_n]/\langle \boldsymbol{t}_0\rangle$.

By differentiating the equality $\boldsymbol{f}_0 = \sigma(B)\ \boldsymbol{t}_0 \in \mathcal{M}_{n,1}(R/(p)[X_1,\ ...,\ X_n])$, we get $\mathrm{Jac}(\boldsymbol{f}_0) = \sigma(B)\ \mathrm{Jac}(\boldsymbol{t}_0)$ in $\mathcal{M}_n(A_0)$. Since $\mathrm{Jac}(\boldsymbol{f}_0)$ is invertible in $\mathcal{M}_n(A_0)$ by hypothesis, B_0 and $\mathrm{Jac}(\boldsymbol{t}_0)$ are invertible in $\mathcal{M}_n(A_0)$ and

$$\sigma(B) = \mathrm{Jac}(\boldsymbol{f}_0)\,\mathrm{Jac}(\boldsymbol{t}_0)^{-1} \tag{6.3}$$

in $\mathcal{M}_n(A_0)$. Because its zeroth p-adic coefficient is invertible, we deduce that $\sigma(B)$ is invertible in $\mathcal{M}_n(R_p[X_1,...,X_n]/\langle \boldsymbol{t}_0\rangle)$. After inverting $\sigma(B)$ in Equation (6.2), it remains to notice that $\boldsymbol{t}-\boldsymbol{t}_0$ is the remainder of $\boldsymbol{t}$ by $\boldsymbol{t}_0$ to conclude. □

Let us explain the idea behind our algorithm. If one takes the coefficient in p^m of Equation (6.1) for $m \geqslant 1$, the left-hand side is the p-adic coefficient $\boldsymbol{t}_m := (t_{1,m},...,t_{n,m})$ of $\boldsymbol{t}$ but the right-hand side depends only on the p-adic coefficients B_i and $\boldsymbol{t}_i$ with $i<m$. Since the matrix B is made of quotients of $\boldsymbol{f}$ by the triangular basis $\boldsymbol{t}$, its coefficient B_i only depends on the coefficients $\boldsymbol{t}_j$ with $j \leqslant i$. So we can deduce $\boldsymbol{t}_m$ from the previous p-adic coefficients of $\boldsymbol{t}$, and compute $\boldsymbol{t}$ at any precision. Informally speaking, we have introduced a shift in the p-adic coefficients of $\boldsymbol{t}$ in the right-hand side.

Computation of $\boldsymbol{B}$ modulo $\boldsymbol{t_0}$ In this paragraph, we explain how to compute the remainder r_i and quotients $\boldsymbol{b}_i$ by $\boldsymbol{t}$ of any element P_i of the result sequence. Since Equation (6.1) is modulo $\boldsymbol{t}_0$, this quantities are only required modulo $\boldsymbol{t}_0$. We proceed recursively on the index i for $-n \leqslant i \leqslant L$.

First, for $-n < i \leqslant 0$, let $\bar{\imath} := i+n$ such that $P_i = X_{\bar{\imath}}$. We distinguish two cases :

- if $\deg_{X_{\bar{\imath}}}(t_{\bar{\imath}}) = 1$, then $\boldsymbol{b}_i := (0,...,0,1,0,...,0)$ with only a one in position $\bar{\imath}$ and $r_i = t_{\bar{\imath}} - X_{\bar{\imath}}$.
- if $\deg_{X_{\bar{\imath}}}(t_{\bar{\imath}}) > 1$, then $X_{\bar{\imath}}$ is already reduced modulo $\boldsymbol{t}$, we put $r_i := X_{\bar{\imath}}$ and $\boldsymbol{b}_i := (0,...,0)$.

Secondly, if the i-th result P_i is a constant in $A/\langle \boldsymbol{t}_0\rangle$, then it is reduced modulo $\boldsymbol{t}$ because $\deg_{X_i}(\sigma(t_i)) = \deg_{X_i}(t_i)$ for any $1 \leqslant i \leqslant n$. Consequently, we take $r_i := P_i$ and $\boldsymbol{b}_i := (0,...,0)$.

Let us consider the final case when $P_i = P_j \,\mathrm{op}\, P_k$ with $\mathrm{op} \in \{+,-,*\}$ and $j,k<i$. The case where op is the addition is straightforward

$$\begin{aligned} r_i &:= r_j + r_k \\ \boldsymbol{b}_i &:= \boldsymbol{b}_j + \boldsymbol{b}_k. \end{aligned}$$

The case of the subtraction is similar. Let us deal with the case of the multiplication. Let

$$s, \boldsymbol{q} := \texttt{Rem\_quo\_triangular}(\boldsymbol{t}, \boldsymbol{t}_0, r_j r_k)$$

be the reductions modulo $\boldsymbol{t}_0$ of the canonical remainder and quotients of $r_j r_k$ by $\boldsymbol{t}$. They satisfy

$$\begin{aligned} r_j r_k &= s && \text{in } A/\langle \boldsymbol{t}\rangle \\ r_j r_k &= s + \boldsymbol{q}\cdot\boldsymbol{t} && \text{in } A/\langle \boldsymbol{t}_0\rangle. \end{aligned}$$

Then one has in $A/\langle \boldsymbol{t}_0 \rangle$

$$\begin{aligned} P_j P_k &= r_j r_k + [(r_j + \boldsymbol{b}_j \cdot \boldsymbol{t}) \times \boldsymbol{b}_k + r_k \times \boldsymbol{b}_j] \cdot \boldsymbol{t} \\ &= s + [\boldsymbol{q} + (r_j + \boldsymbol{b}_j \cdot \boldsymbol{t}) \times \boldsymbol{b}_k + r_k \times \boldsymbol{b}_j] \cdot \boldsymbol{t} \end{aligned}$$

which implies, still in $A/\langle \boldsymbol{t}_0 \rangle$,

$$\begin{aligned} r_i &:= s \\ \boldsymbol{b}_i &:= \boldsymbol{q} + (r_j + \boldsymbol{b}_j \cdot \boldsymbol{t}) \times \boldsymbol{b}_k + r_k \times \boldsymbol{b}_j. \end{aligned} \tag{6.4}$$

We put all these formulas together to form an algorithm that computes all the remainders r_i and quotients $\boldsymbol{b}_i$ modulo $\boldsymbol{t}_0$. We describe this algorithm as a straight-line program, in order to prove that it is a part of a shifted algorithm.

Let L be the length of the s.l.p. Γ of $\boldsymbol{f}$. We define recursively in i such that $-n < i \leqslant L$ some s.l.p.'s ε^i with n inputs. These s.l.p.'s ε^i compute, on the entries $\boldsymbol{t}$ given as the list of their polynomial coefficients, the remainders r_j and quotients $\boldsymbol{b}_j$ of P_j for $j < i$. We call ρ^i and $\boldsymbol{\alpha}^i = (\alpha^i_1, ..., \alpha^i_n)$ the outputs of ε^i corresponding to r_i and $\boldsymbol{b}_i$.

Definition 6.15. *Let us initiate the induction for $-n < i \leqslant 0$ and $\bar{\imath} := i + n$:*

- *if $\deg_{X_{\bar{\imath}}}(t_{\bar{\imath}}) = 1$, then we define $\varepsilon^i := (-r_{\bar{\imath}}, 0, 1)$ where $r_{\bar{\imath}} := t_{\bar{\imath}} - X_{\bar{\imath}}$. The output ρ^i points to $-r_{\bar{\imath}}$ and α^i_m points to 0 if $m \neq \bar{\imath}$ or 1 otherwise;*
- *if $\deg_{X_{\bar{\imath}}}(t_{\bar{\imath}}) > 1$, then we define $\varepsilon^i := (X_{\bar{\imath}}, 0)$. The output ρ^i points to $X_{\bar{\imath}}$ and α^i_m points to 0 for any $1 \leqslant m \leqslant n$.*

Now recursively for $0 < i \leqslant L$, depending on the operation type of Γ_i:

- *if $\Gamma_i = (P^c)$ with $P \in A$ reduced modulo $\boldsymbol{t}_0$, then we define $\varepsilon^i := (P, 0)$. The output ρ^i points to P and α^i_m points to 0 for any $1 \leqslant m \leqslant n$;*
- *if $\Gamma_i = (+; u, v)$, then we build ε^i on top of ε^u and ε^v in such a manner that one has $\rho^i := \rho^u + \rho^v$ and $\boldsymbol{\alpha}^i := \boldsymbol{\alpha}^u + \boldsymbol{\alpha}^v$;*
- *if $\Gamma_i = (-; u, v)$, then we build ε^i on top of ε^u and ε^v in such a manner that one has $\rho^i := \rho^u - \rho^v$ and $\boldsymbol{\alpha}^i := \boldsymbol{\alpha}^u - \boldsymbol{\alpha}^v$;*
- *if $\Gamma_i = (*; u, v)$, we define ε^i accordingly to formula (6.4). First, we compute $s, \boldsymbol{q} := \texttt{Rem\_quo\_triangular}(\rho^u(\boldsymbol{t})\ \rho^v(\boldsymbol{t}), \boldsymbol{t}, \boldsymbol{t}_0)$. Then $\rho^i := s$ and $\boldsymbol{\alpha}^i$ is defined by*

$$\boldsymbol{q} + (\rho^u(\boldsymbol{t}) + \boldsymbol{\alpha}^u(\boldsymbol{t}) \cdot \boldsymbol{t}) \times \boldsymbol{\alpha}^v(\boldsymbol{t}) + \rho^v(\boldsymbol{t}) \times \boldsymbol{\alpha}^u(\boldsymbol{t}).$$

Finally, we set $\varepsilon = \varepsilon^L$.

Shifted algorithm In this paragraph, we prove that formula (6.1) gives rise to a shifted algorithm to compute $\boldsymbol{t}$. Mainly, we have to prove that the p-adic coefficient in p^m of $p^2\,(\delta(B) \cdot \delta(\boldsymbol{t}))$, that is the coefficient in p^{m-2} of $\delta(B) \cdot \delta(\boldsymbol{t})$, depend at most in the coefficients $\boldsymbol{t}_i$ of $\boldsymbol{t}$ with $i < m$. For that matter, we will compute the shift index of the computation of $p^2\,(\delta(B) \cdot \delta(\boldsymbol{t}))$ and prove that it is positive.

Since the s.l.p. ε computes the matrix B on the entries $\boldsymbol{t}$, we can build an s.l.p. $\boldsymbol{\Lambda}$ on top of ε such that

$$\boldsymbol{\Lambda}: \boldsymbol{t} \mapsto \boldsymbol{t}_0 + [\sigma(B)^{-1} (\boldsymbol{f} - p^2 \times (\delta(B) \cdot \delta(\boldsymbol{t}))) \text{ rem } \boldsymbol{t}_0].$$

In the s.l.p. $\boldsymbol{\Lambda}$, the resolution of the linear system

$$\sigma(B)\, \boldsymbol{a} = (\boldsymbol{f} - p^2 \times (\delta(B) \cdot \delta(\boldsymbol{t}))) \in \mathcal{M}_{n,1}(A/\langle \boldsymbol{t}_0 \rangle)$$

in $\boldsymbol{a}$ is performed by the relaxed algorithm of Chapter 3, Section 3.3.2. Indeed, $\sigma(B)$ has length 1 and this algorithm is adapted to low length matrices.

Lemma 6.16. *The s.l.p. $\boldsymbol{\Lambda}$ is a shifted algorithm of which $\boldsymbol{t}$ is a fixed point when the computations are done in the algebra $R_p[X_1, ..., X_n]$.*

Proof. The triangular set $\boldsymbol{t}$ is a fixed point of the s.l.p. $\boldsymbol{\Lambda}$ over $R_p[X_1, ..., X_n]$ because of Equation (6.1).

Since the s.l.p. ε uses only additions, subtractions, multiplications, calls to `Rem_triangular` and `Rem_quo_triangular`, and since all these operations preserve a non-negative shift index (Lemma 6.9), we know that $\mathrm{sh}(\boldsymbol{t} \mapsto B) \geqslant 0$. Besides

$$\begin{aligned} \mathrm{sh}(\boldsymbol{t} \mapsto p^2 \times (\delta(B) \cdot \delta(\boldsymbol{t}))) &= 2 + \mathrm{sh}(\boldsymbol{t} \mapsto \delta(B) \cdot \delta(\boldsymbol{t})) \\ &= 2 + \min(\mathrm{sh}(\boldsymbol{t} \mapsto \delta(B)), \mathrm{sh}(\boldsymbol{t} \mapsto \delta(\boldsymbol{t}))) \\ &= 1 + \min(\mathrm{sh}(\boldsymbol{t} \mapsto B), \mathrm{sh}(\boldsymbol{t} \mapsto \boldsymbol{t})) \\ &\geqslant 1. \end{aligned}$$

Furthermore, notice that $\boldsymbol{f} \operatorname{rem} \boldsymbol{t}_0$ and $\sigma(B)$ depend only on $\boldsymbol{t}_0$. Finally the resolution of the linear system does not change the shift, hence we have proved that $\mathrm{sh}(\boldsymbol{\Lambda}) > 0$. □

Proof. (of Theorem 6.2) The triangular set $\boldsymbol{t}$ is a fixed point of the s.l.p. $\boldsymbol{\Lambda}$, which is a shifted algorithm by Lemma 6.16. Proposition 2.17 shows that we can compute $\boldsymbol{t}$ in time the number of operations in $\boldsymbol{\Lambda}$.

We count the number of operations of $\boldsymbol{\Lambda}$:

- Computation of the remainder r and the quotients $\boldsymbol{b}$ at each step of the computation of $\boldsymbol{f}$:

  We focus on the steps which correspond to a multiplication $*$ in the s.l.p. $\boldsymbol{f}$ because they have the worst complexity. The remainder and quotients require a call to Algorithm `Rem_quo_triangular`. Then $\boldsymbol{b}$ uses an inner product and scalar vector multiplications $\times$. The inner product costs less than a call to `Rem_quo_triangular`, since this latter algorithm does an inner product. Summing up, the total cost is

  $$\mathcal{O}(L\, \mathsf{R}(N)\, \mathsf{RemQuo}(d_1, ..., d_n) + n\, L\, \mathsf{R}(N)\, \mathsf{Rem}(d_1, ..., d_n))$$

  that is $\mathcal{O}(n\, L\, \mathsf{R}(N)\, \mathsf{Rem}(d_1, ..., d_n))$ (see Proposition 6.8);
- Computation of $\boldsymbol{f} \operatorname{rem} \boldsymbol{t}_0$ in time $\mathcal{O}(L\, \mathsf{Rem}(d_1, ..., d_n)\, \mathsf{R}(N))$;
- Computation of $p^2 \times (\delta(B) \cdot \delta(\boldsymbol{t}))$ requires n inner products $\delta(b_i) \cdot \delta(\boldsymbol{t})$, whose costs are dominated by $\mathcal{O}(n\, \mathsf{R}(N)\, \mathsf{RemQuo}(d_1, ..., d_n))$, which is bounded by $\mathcal{O}(n\, L\, \mathsf{R}(N)\, \mathsf{Rem}(d_1, ..., d_n))$ since $L \geqslant n$;

- Resolution of the linear system in $\sigma(B)$:
  Since $\sigma(B)$ has length one, Proposition 3.6 solves the linear system in time $\mathcal{O}(N\,\mathsf{MMR}(n,1,1)/1+n^\Omega)=\tilde{\mathcal{O}}(n^2)\,N+\mathcal{O}(n^\Omega)$. □

6.4.2 By-passing the whole quotient matrix

In the algorithm of Section 6.4.1, we computed the whole quotient B. This raised a component $\mathcal{O}(n\,L\,\mathsf{R}(N)\,\mathsf{Rem}(d_1,\ldots,d_n))$ in the complexity. We also had to call `Rem_quo_triangular` for each multiplication in the s.l.p. of $\boldsymbol{f}$, leading to a cost of $\mathcal{O}(e\,L\,\mathsf{R}(N)\,\mathsf{Rem}(d_1,\ldots,d_n))$. These two costs are balanced when $e\simeq n$.

However, when $e\ll n$, we can benefit from not computing the whole quotient B. We present in this section a new method to compute $\delta(B)\cdot\delta(\boldsymbol{t})$ without computing B, thus leading to an asymptotic complexity of $\mathcal{O}(L\,\mathsf{R}(N)\,\mathsf{Rem}(d_1,\ldots,d_n))$ plus some calls to `Rem_quo_triangular`. That is how we reach a total complexity of $\mathcal{O}(e\,L\,\mathsf{R}(N)\,\mathsf{Rem}(d_1,\ldots,d_n))$.

Nevertheless, this new method makes it harder to deal with the carries involved in the computation of B. We introduce the notion of shifted decomposition to solve this issue. In return, we increase the subdominant part of the complexity when N tends to infinity.

Shifted decomposition Recall that σ and δ were defined by $\sigma(a)=a_0$ and $\delta(a):=\frac{a-a_0}{p}$ and that, for any $a\in R_p$, we have $a=a_0+p\,\delta(a)$.

To our great regret, σ and δ are not ring homomorphisms. To remedy this fact, we call a *shifted decomposition* of $a\in R_p$ a pair $(\sigma_a,\,\delta_a)\in R_p^2$ such that $a=\sigma_a+p\,\delta_a$. Shifted decompositions are not unique. For any $a\in R_p$, the pair $(\sigma(a),\delta(a))$ is called the *canonical* shifted decomposition of a. Because σ and δ are not ring homomorphisms, we will use another shifted decomposition that behaves better with respect to arithmetic operations.

Lemma 6.17. *Let $a,b\in R_p$ and $(\sigma_a,\delta_a),(\sigma_b,\delta_b)\in R_p^2$ be shifted decompositions of a and b. Then*

1. *$(\sigma_a+\sigma_b,\delta_a+\delta_b)$ is a shifted decomposition of $a+b$;*
2. *$(\sigma_a-\sigma_b,\delta_a-\delta_b)$ is a shifted decomposition of $a-b$;*
3. *$(\sigma_a\,\sigma_b,\delta_a\,\sigma_b+a\,\delta_b)$ and $(\sigma_a\,\sigma_b,\delta_a\,b+\sigma_a\,\delta_b)$ are shifted decompositions of $a\,b$.*

Proof. These shifted decompositions are direct consequences of the relations

$$a+b = \sigma_a+\sigma_b+p\,(\delta_a+\delta_b) \tag{6.5}$$

$$-a = -\sigma_a+p\,(-\delta_a) \tag{6.6}$$

$$\begin{aligned} a\,b &= \sigma_a\,\sigma_b+p\,(\delta_a\,b+\sigma_a\,\delta_b)\\ &= \sigma_a\,\sigma_b+p\,(\delta_a\,\sigma_b+a\,\delta_b). \end{aligned} \tag{6.7}$$

□

We extend the notion of shifted decomposition naturally to polynomials $R_p[X_1, ..., X_n]$, vectors $(R_p)^n$ and matrices $\mathcal{M}_{r,s}(R_p)$.

Recursive formula for $\boldsymbol{t}$ The recursive formula (6.1) for $\boldsymbol{t}$ adapts well to shifted decomposition.

Lemma 6.18. *Let (σ_B, δ_B) be any shifted decomposition of the quotient matrix $B \in \mathcal{M}_n(R_p[X_1, ..., X_n])$.*

Then the matrix $\sigma_B \in \mathcal{M}_n(R_p[X_1, ..., X_n])$ is invertible modulo $\boldsymbol{t}_0$. Moreover the triangular set $\boldsymbol{t}$ satisfies the recursive equation

$$\boldsymbol{t} - \boldsymbol{t}_0 = \sigma_B^{-1} \left(\boldsymbol{f} - p^2 \left(\delta_B \cdot \delta(\boldsymbol{t}) \right) \right) \text{ rem } \boldsymbol{t}_0 \tag{6.8}$$

in $\mathcal{M}_{n,1}(R_p[X_1, ..., X_n])$.

Proof. We proceed similarly to the proof of Lemma 6.14. For any $P \in R[X_1, ..., X_n]$, let r and $\boldsymbol{a}$ be the canonical remainder and quotients of P by $\boldsymbol{t}$. For any shifted decomposition $(\sigma_{\boldsymbol{a}}, \delta_{\boldsymbol{a}})$ of $\boldsymbol{a}$, one has

$$P - r = \boldsymbol{a} \cdot \boldsymbol{t} = \sigma_{\boldsymbol{a}} \cdot \boldsymbol{t} + p\, \delta_{\boldsymbol{a}} \cdot \boldsymbol{t} = \sigma_{\boldsymbol{a}} \cdot \boldsymbol{t} + p\, \delta_{\boldsymbol{a}} \cdot \boldsymbol{t}_0 + p^2\, \delta_{\boldsymbol{a}} \cdot \delta(\boldsymbol{t}).$$

We apply this to the equations $\boldsymbol{f}$ and get

$$\sigma_B \cdot \boldsymbol{t} = \boldsymbol{f} - p^2 \left(\delta_B \cdot \delta(\boldsymbol{t}) \right)$$

in $R_p[X_1, ..., X_n]/\langle \boldsymbol{t}_0 \rangle$.

Since the zeroth p-adic coefficient of σ_B is the one of B which in invertible in $\mathcal{M}_n(A_0)$ (see the proof of Lemma 6.14), we deduce that σ_B is invertible in $\mathcal{M}_n(R_p[X_1, ..., X_n]/\langle \boldsymbol{t}_0 \rangle)$. It remains to invert σ_B and to notice that $\boldsymbol{t} - \boldsymbol{t}_0 = \boldsymbol{t}$ rem $\boldsymbol{t}_0$ to conclude. □

Computation of $\boldsymbol{r}$, $\boldsymbol{\sigma_B}$ and $\boldsymbol{\delta_B \cdot \delta(t)}$ For every multiplication of the s.l.p. Γ of $\boldsymbol{f}$, we did n calls to `Rem_triangular` and one call to `Rem_quo_triangular` to compute the corresponding quotients with our first method of subsection 6.4.1. In this paragraph, we present a method that does only $\mathcal{O}(1)$ calls to `Rem_triangular` and one call to `Rem_quo_triangular` in the same situation.

We denote by $(\sigma_{\boldsymbol{b}_i}, \delta_{\boldsymbol{b}_i})$ a shifted decomposition of the quotients $\boldsymbol{b}_i$. The main idea of our new method is to deal with $\delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t}) \in R_p$ instead of $\delta_{\boldsymbol{b}_i} \in (R_p)^n$. Let us explain how to compute r_i, $\sigma_{\boldsymbol{b}_i}$ and $\delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t})$. We proceed recursively on the index i for $-n < i \leqslant L$.

First, for an index i corresponding to an input, *i.e.* $-n < i \leqslant 0$, we set $\bar{\imath} := i + n$. Therefore $P_i = X_{\bar{\imath}}$ and we distinguish two cases:

- if $\deg_{X_{\bar{\imath}}}(t_{\bar{\imath}}) = 1$, then we set $r_i := t_{\bar{\imath}} - X_{\bar{\imath}} \in R_p[X_1, ..., X_{i-1}]$ reduced with respect to $t_1, ..., t_{i-1}$. Also we set $\sigma_{\boldsymbol{b}_i} := (0, ..., 0, 1, 0, ..., 0)$ the vector with only a one at position $\bar{\imath}$ and $\delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t}) = 0$.
- if $\deg_{X_{\bar{\imath}}}(t_{\bar{\imath}}) > 1$, then $X_{\bar{\imath}}$ is already reduced modulo $\boldsymbol{t}$ and we take

$$r_i := X_{\bar{\imath}}, \quad \sigma_{\boldsymbol{b}_i} := (0, ..., 0), \quad \delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t}) = 0. \tag{6.9}$$

Now let $0 < i \leqslant L$ that corresponds to operations in Γ. If the i-th result P_i is a constant in $A/\langle \boldsymbol{t}_0 \rangle$, then, as before, we take

$$r_i := P_i, \quad \sigma_{\boldsymbol{b}_i} := (0, ..., 0), \quad \delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t}) = 0. \tag{6.10}$$

Consider the final case when $P_i = P_j \,\mathrm{op}\, P_k$ with $\mathrm{op} \in \{+, -, *\}$ and $j, k < i$. The case where op is the addition is straightforward; using Lemma 6.17, one takes

$$\begin{aligned} r_i &:= r_j + r_k \\ \sigma_{\boldsymbol{b}_i} &:= \sigma_{\boldsymbol{b}_j} + \sigma_{\boldsymbol{b}_k} \\ \delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t}) &:= \delta_{\boldsymbol{b}_j} \cdot \delta(\boldsymbol{t}) + \delta_{\boldsymbol{b}_k} \cdot \delta(\boldsymbol{t}). \end{aligned} \tag{6.11}$$

The case of subtraction is similar. Let us deal with the more complicated case of multiplication. We start by computing the remainder and quotients

$$s, \boldsymbol{q} := \texttt{Rem\_quo\_triangular}(\boldsymbol{t}, \boldsymbol{t}_0, r_j r_k)$$

of $r_j r_k$ by $\boldsymbol{t}$ modulo $\boldsymbol{t}_0$. They satisfy

$$\begin{aligned} r_j r_k &= s && \text{in } A/\langle \boldsymbol{t} \rangle \\ r_j r_k &= s + \boldsymbol{q} \cdot \boldsymbol{t} && \text{in } A/\langle \boldsymbol{t}_0 \rangle. \end{aligned}$$

Thus we still have over $A/\langle \boldsymbol{t}_0 \rangle$

$$\begin{aligned} r_i &:= s \\ \boldsymbol{b}_i &:= \boldsymbol{q} + (r_j + \boldsymbol{b}_j \cdot \boldsymbol{t}) \times \boldsymbol{b}_k + r_k \times \boldsymbol{b}_j. \end{aligned} \tag{6.12}$$

We use the formulas of Lemma 6.17 to compute the shifted decomposition of $\boldsymbol{b}_i$ from shifted decompositions of its operands. Shifted decompositions of $\boldsymbol{b}_j$ and $\boldsymbol{b}_k$ were computed at a previous step of the recursion. We choose to take the canonical shifted decomposition for $r_j, r_k, \boldsymbol{q}$ and $\boldsymbol{t}$. Since the scalar multiplication operator $\times$ and the inner product $\cdot$ are made of additions and multiplications, we deduce that we can take

$$\begin{aligned} \sigma_{\boldsymbol{b}_i} &= \boldsymbol{q}_0 + ((r_j)_0 + \sigma_{\boldsymbol{b}_j} \cdot \boldsymbol{t}_0) \times \sigma_{\boldsymbol{b}_k} + (r_k)_0 \times \sigma_{\boldsymbol{b}_j} \\ \delta_{\boldsymbol{b}_i} &= \delta(\boldsymbol{q}) + (\delta(r_j) + \delta_{\boldsymbol{b}_j} \cdot \boldsymbol{t}_0 + \boldsymbol{b}_j \cdot \delta(\boldsymbol{t})) \times \boldsymbol{b}_k + ((r_j)_0 + \sigma_{\boldsymbol{b}_j} \cdot \boldsymbol{t}_0) \times \delta_{\boldsymbol{b}_k} + \delta(r_k) \times \boldsymbol{b}_j + \\ &\quad (r_k)_0 \times \delta_{\boldsymbol{b}_j}. \end{aligned}$$

Because we work in $A/\langle \boldsymbol{t}_0 \rangle$, this decomposition simplifies and we define

$$\begin{aligned} \sigma_{\boldsymbol{b}_i} &:= \boldsymbol{q}_0 + (r_j)_0 \times \sigma_{\boldsymbol{b}_k} + (r_k)_0 \times \sigma_{\boldsymbol{b}_j} \\ \delta_{\boldsymbol{b}_i} &:= \delta(\boldsymbol{q}) + \delta(r_k) \times \boldsymbol{b}_j + (r_k)_0 \times \delta_{\boldsymbol{b}_j} + \\ &\quad (\delta(r_j) + \boldsymbol{b}_j \cdot \delta(\boldsymbol{t})) \times \boldsymbol{b}_k + (r_j)_0 \times \delta_{\boldsymbol{b}_k}. \end{aligned} \tag{6.13}$$

Now that we have computed r_i and $\sigma_{\boldsymbol{a}_i}$, it remains to compute $\delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t})$. Using $\sigma_{\boldsymbol{b}_j}$, $\sigma_{\boldsymbol{b}_k}, \delta_{\boldsymbol{b}_j} \cdot \delta(\boldsymbol{t}), \delta_{\boldsymbol{b}_k} \cdot \delta(\boldsymbol{t})$ and other known polynomials, we compute

$$\begin{aligned} \delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t}) &:= \delta(\boldsymbol{q}) \cdot \delta(\boldsymbol{t}) + \delta(r_k)\,(\boldsymbol{b}_j \cdot \delta(\boldsymbol{t})) + (r_k)_0\,(\delta_{\boldsymbol{b}_j} \cdot \delta(\boldsymbol{t})) + \\ &\quad (\delta(r_j) + \boldsymbol{b}_j \cdot \delta(\boldsymbol{t}))\,(\boldsymbol{b}_k \cdot \delta(\boldsymbol{t})) + (r_j)_0\,(\delta_{\boldsymbol{b}_k} \cdot \delta(\boldsymbol{t})) \end{aligned} \tag{6.14}$$

where $\boldsymbol{b}_j \cdot \delta(\boldsymbol{t}) := \sigma_{\boldsymbol{b}_j} \cdot \delta(\boldsymbol{t}) + p\,(\delta_{\boldsymbol{b}_j} \cdot \delta(\boldsymbol{t}))$ and the same for $\boldsymbol{b}_k \cdot \delta(\boldsymbol{t})$. This formula is new and admits no equivalents for canonical shifted decompositions when the p-adics have carries.

We sum up all these computations in an algorithm. We define recursively for $-n < i \leqslant L$ some s.l.p.'s ξ^i with n inputs. These s.l.p.'s ξ^i compute, on the entries $\boldsymbol{t}$ given as the list of their polynomial coefficients, the remainder r_j and the quantities $\sigma_{\boldsymbol{b}_j}$ and $\delta_{\boldsymbol{b}_j} \cdot \delta(\boldsymbol{t})$ for $j < i$. We name ρ^i, $\boldsymbol{\alpha}^i = (\alpha_1^i, ..., \alpha_n^i)$ and θ^i the outputs of ξ^i corresponding to r_i, $\sigma_{\boldsymbol{b}_i}$ and $\delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t})$.

Definition 6.19. *Let us initiate the induction for $-n < i \leqslant 0$ and $\bar{\imath} := i + n$:*

- *if $\deg_{X_{\bar{\imath}}}(t_{\bar{\imath}}) = 1$, then we define $\xi^i := (-r_{\bar{\imath}}, 0, 1)$ where $r_{\bar{\imath}} := t_{\bar{\imath}} - X_{\bar{\imath}}$. The output ρ^i points to $-r_{\bar{\imath}}$, α_m^i points to 0 if $m \neq \bar{\imath}$ or 1 otherwise and θ^i points to 0;*
- *if $\deg_{X_{\bar{\imath}}}(t_{\bar{\imath}}) > 1$, then we define $\xi^i := (X_{\bar{\imath}}, 0)$. The output ρ^i points to $X_{\bar{\imath}}$, θ^i and α_m^i points to 0 for any $1 \leqslant m \leqslant n$.*

Now recursively for $0 < i \leqslant L$, depending on the operation type of Γ_i:

- *if $\Gamma_i = (P^c)$ with $P \in A$ reduced modulo $\boldsymbol{t}_0$, then we define $\xi^i := (P, 0)$. The output ρ^i points to P and α_m^i points to 0 for any $1 \leqslant m \leqslant n$;*
- *if $\Gamma_i = (+; u, v)$, then we build ξ^i on top of ξ^u and ξ^v in such a manner that one has $\rho^i := \rho^u + \rho^v$, $\boldsymbol{\alpha}^i := \boldsymbol{\alpha}^u + \boldsymbol{\alpha}^v$ and $\theta^i := \theta^u + \theta^v$;*
- *if $\Gamma_i = (-; u, v)$, then we build ξ^i on top of ξ^u and ξ^v in such a manner that one has $\rho^i := \rho^u - \rho^v$, $\boldsymbol{\alpha}^i := \boldsymbol{\alpha}^u - \boldsymbol{\alpha}^v$ and $\theta^i := \theta^u - \theta^v$;*
- *if $\Gamma_i = (*; u, v)$, we define ξ^i accordingly to formulas (6.12,6.13,6.14). First, we compute $s, \boldsymbol{q} := \texttt{Rem\_quo\_triangular}(\rho^u(\boldsymbol{t})\,\rho^v(\boldsymbol{t}), \boldsymbol{t}, \boldsymbol{t}_0)$. Then $\rho^i := s$, $\boldsymbol{\alpha}^i$ is defined by*

$$\sigma(\boldsymbol{q}) + (\rho^u(\boldsymbol{t}) + \boldsymbol{\alpha}^u(\boldsymbol{t}) \cdot \boldsymbol{t}) \times \boldsymbol{\alpha}^v(\boldsymbol{t}) + \rho^v(\boldsymbol{t}) \times \boldsymbol{\alpha}^u(\boldsymbol{t})$$

and θ^i is defined by

$$\delta(\boldsymbol{q}) \cdot \delta(\boldsymbol{t}) + \delta(\rho^v)\,(\Theta^u) + (\rho^v)_0\,(\theta^u) + (\delta(\rho^u) + \Theta^u)\,(\Theta^v) + (\rho^u)_0\,(\theta^v)$$

where $\Theta^u := \boldsymbol{\alpha}^u \cdot \delta(\boldsymbol{t}) + p \times \theta^u$ and the same for Θ^v.

Finally, we set $\xi = \xi^L$.

Shifted algorithm Similarly to Section 6.4.1, we prove that Lemma 6.18 gives rise to a shifted algorithm to compute $\boldsymbol{t}$. For that matter, we will compute the shift index of the computation of $p^2\,(\delta_B \cdot \delta(\boldsymbol{t}))$ and prove that it is positive.

Lemma 6.20. *For any $-n < i \leqslant L$, one has*

$$\mathrm{sh}(\boldsymbol{t} \mapsto \rho^i(\boldsymbol{t})) \geqslant 0, \quad \mathrm{sh}(\boldsymbol{t} \mapsto \boldsymbol{\alpha}^i(\boldsymbol{t})) \geqslant 0, \quad \mathrm{sh}(\boldsymbol{t} \mapsto \theta^i(\boldsymbol{t})) \geqslant -1.$$

Proof. We proceed recursively on i for $-n < i \leqslant L$.

We initialize the induction for any $-n < i \leqslant 0$. One has

$$\mathrm{sh}(\boldsymbol{t} \mapsto \boldsymbol{\alpha}^i(\boldsymbol{t})) = \mathrm{sh}(\boldsymbol{t} \mapsto \theta^i(\boldsymbol{t})) = +\infty, \quad \mathrm{sh}(\boldsymbol{t} \mapsto \rho^i(\boldsymbol{t})) = \begin{cases} +\infty & \text{if } \deg_{X_{\bar{\imath}}}(t_{\bar{\imath}}) > 1 \\ 0 & \text{otherwise} \end{cases}.$$

Now recursively for $0 < i \leqslant L$, depending on the type of the i-th operation of Γ:

- if $\Gamma_i = (P^c)$ with $P \in A$ reduced modulo $\boldsymbol{t}_0$, one has
$$\mathrm{sh}(\boldsymbol{t} \mapsto \rho^i(\boldsymbol{t})) = \mathrm{sh}(\boldsymbol{t} \mapsto \boldsymbol{\alpha}^i(\boldsymbol{t})) = \mathrm{sh}(\boldsymbol{t} \mapsto \theta^i(\boldsymbol{t})) = +\infty.$$
- if $\Gamma_i = (\omega; u, v)$ with $\omega \in \{+, -, *\}$ then we proceed as follows. The s.l.p. ξ^i uses only additions, subtractions, multiplications, shifts $p \times \_$ by p, and calls to `Rem_triangular` and `Rem_quo_triangular`. These operations preserve a non-negative shift index, so
$$\mathrm{sh}(\boldsymbol{t} \mapsto \rho^i(\boldsymbol{t})) \geqslant 0, \quad \mathrm{sh}(\boldsymbol{t} \mapsto \boldsymbol{\alpha}^i(\boldsymbol{t})) \geqslant 0.$$
Now θ^i is an arithmetic expression in $\delta(\boldsymbol{q})$, $\delta(\boldsymbol{t})$, $(\rho^u)_0$, $\delta(\rho^u)$, $\boldsymbol{\alpha}^u$, θ^u, $p \times \theta^u$ and the same for v. All this quantities have a shift index greater or equal to -1 and so it is for θ^i. □

Since the s.l.p. ξ computes on the entries $\boldsymbol{t}$ the p-adic vector $\delta_B \cdot \delta(\boldsymbol{t})$, we can build an s.l.p. $\boldsymbol{\Delta}$ on top of ξ such that

$$\boldsymbol{\Delta}: \boldsymbol{t} \mapsto \boldsymbol{t}_0 + [\sigma_B^{-1}\,(\boldsymbol{f} - p^2 \times (\delta_B \cdot \delta(\boldsymbol{t}))) \text{ rem } \boldsymbol{t}_0].$$

The resolution of the linear system in $\boldsymbol{\Delta}$ is done by the relaxed algorithm of Chapter 3, Section 3.3.3.

Lemma 6.21. *The s.l.p. $\boldsymbol{\Delta}$ is a shifted algorithm of which $\boldsymbol{t}$ is a fixed point when the computations are done in the algebra $R_p[X_1, ..., X_n]$.*

Proof. The s.l.p. $\boldsymbol{\Delta}$ compute $\boldsymbol{t}$ on the entries $\boldsymbol{t}$ in the algebra $R_p[X_1, ..., X_n]$ thanks to Lemma 6.18 and because the formulas that define ξ in Definition 6.19 match formulas (6.9) to (6.14).

A direct consequence of Lemma 6.20 is that

$$\mathrm{sh}(\boldsymbol{t} \mapsto p^2 \times (\delta_B \cdot \delta(\boldsymbol{t}))) \geqslant (2 + \mathrm{sh}(\boldsymbol{t} \mapsto \delta_B \cdot \delta(\boldsymbol{t}))) \geqslant 1.$$

Since $\boldsymbol{f} \operatorname{rem} \boldsymbol{t}_0$ and σ_B depend only on $\boldsymbol{t}_0$, and since the resolution of the linear system does not impact the shift, we have proved $\boldsymbol{\Delta}$ has a positive shift index. □

Proof. (of Theorem 6.3) By Lemma 6.21, the triangular set $\boldsymbol{t}$ is a fixed point of the shifted algorithm $\boldsymbol{\Delta}$. Proposition 2.17 shows that we can compute $\boldsymbol{t}$ in time the number of operations in $\boldsymbol{\Delta}$. Let us count the number of operations in $\boldsymbol{\Delta}$.

Cost of $\boldsymbol{\sigma_B}$. We start by evaluating the maximal length of the entries of σ_B. We look at the effect of one operation of the s.l.p. of $\boldsymbol{f}$ on $\sigma_{\boldsymbol{b}_i}$. The worst case happens for the multiplication $*$. In this case, recall from Formula 6.13 that

$$\sigma_{\boldsymbol{b}_i} = \boldsymbol{q}_0 + (r_j)_0 \times \sigma_{\boldsymbol{b}_k} + (r_k)_0 \times \sigma_{\boldsymbol{b}_j}.$$

The multiplication modulo a triangular set increase the length of the p-adics by a factor $\tilde{\mathcal{O}}(d_1 \cdots d_n)$ (see [Lan91, Theorem 3]), so that

$$\lambda(\sigma_{\boldsymbol{b}_i}) \leqslant \max(\lambda(\sigma_{\boldsymbol{b}_k}), \lambda(\sigma_{\boldsymbol{b}_j})) + 2\,\tilde{\mathcal{O}}(d_1 \cdots d_n) + 2\,.$$

There are L operations and consequently $\lambda := \lambda(\sigma_B) = \tilde{\mathcal{O}}(L\,d_1 \cdots d_n)$.

The multiplication of two p-adics of length 1 and λ costs $\mathcal{O}(\min(\lambda, N))$ and so does the addition of two p-adics of length λ. Put it all together for a total cost of $\mathcal{O}(n\,L \min(\lambda, N)\,\mathsf{Rem}(d_1, ..., d_n))$.

Computation of $\boldsymbol{f}$ rem $\boldsymbol{t_0}$ in time $\mathcal{O}(L\,\mathsf{Rem}(d_1, ..., d_n)\,\mathsf{R}(N))$.

Computation of $\boldsymbol{r_i}$ and $\boldsymbol{\delta_{b_i} \cdot \delta(t)}$ at each step of the computation of $\boldsymbol{f}$. We focus on the operations of Γ which are multiplications because they induce the more operations in $\boldsymbol{\Delta}$. The remainder s and quotients $\boldsymbol{q}$ of $r_j\,r_k$ require a call to Algorithm `Rem_quo_triangular`. Then $\delta_B \cdot \delta(\boldsymbol{t})$ use an inner product $\delta(\boldsymbol{q}) \cdot \delta(\boldsymbol{t})$ and $\mathcal{O}(1)$ multiplications in $R_p[X_1, ..., X_n]/\langle \boldsymbol{t}_0 \rangle$. The inner product costs less than a call to `Rem_quo_triangular`, since this latter algorithm does an inner product $\boldsymbol{q} \cdot \boldsymbol{t}$. Summing up, the total cost is $\mathcal{O}(L\,\mathsf{R}(N)\,(\mathsf{RemQuo}(d_1, ..., d_n) + \mathsf{Rem}(d_1, ..., d_n)))$, that is $\mathcal{O}(e\,L\,\mathsf{R}(N)\,\mathsf{Rem}(d_1, ..., d_n))$.

Resolution of the linear system in $\boldsymbol{\sigma_B}$: Since the matrix σ_B has finite length λ, Proposition 3.6 tells us that the cost of solving the linear system is

$$\mathcal{O}([N\,\mathsf{MMR}(n, 1, \lambda)/\lambda + n^{\Omega}]\,\mathsf{Rem}(d_1, ..., d_n)),$$

that is $[N n^{2+o(1)} \log(\lambda)^{\mathcal{O}(1)} + \mathcal{O}(n^{\Omega})]\,\mathsf{Rem}(d_1, ..., d_n)$. □

6.5 Implementation in `Mathemagix`

The computer algebra software MATHEMAGIX [HLM+02] provides a C++ library named ALGEBRAMIX implementing relaxed power series or p-adic numbers [Hoe02, Hoe07, BHL11, BL12]. We implemented the lifting of univariate representations over the power series ring $\mathbb{F}_p[[X]]$ for both the off-line and the relaxed approach. The implementations over the p-adic integers $\mathbb{Z}_p$ are still in progress. Although they work, they still require some efforts to be competitive.

Our implementation is available in the files whose name begins with `lift_` in the C++ library GREGORIX of MATHEMAGIX. We mention that our on-line algorithm is currently connected to KRONECKER inside MATHEMAGIX with the help of G. LECERF.

We now give some implementation details:

- for the multiplication of polynomials of power series in $(\mathbb{F}_p[[X]])[Y]$, we first converted them as power series of polynomial in $(\mathbb{F}_p[Y])[[X]]$. Then the relaxed multiplication algorithm reduces to multiplications of finite precision power series of polynomials, that is polynomials of polynomials in $(\mathbb{F}_p[Y])[X]$. We classically used a Kronecker substitution to reduce these products to multiplications of univariate polynomials in $\mathbb{F}_p[Z]$;
- since remainder and quotients modulo $\boldsymbol{t}$ or $\boldsymbol{t}_0$ are often used, we stored the precomputation of the inverse of these elements in Algorithms `Rem_triangular` and `Rem_quo_triangular`;
- for the matrix multiplication modulo Q inside the off-line Algorithm `Global_Newton_step_univariate`, we delayed the reductions until after the matrix multiplication to reduce their numbers;

- as we mentioned before, Algorithms `Rem_triangular` and `Rem_quo_triangular` greatly simply with univariate representations since we just have to compute quotient and remainder of univariate polynomials.

6.5.1 Benchmarks

We report the timings of our implementation in milliseconds. Timings are measured using one core of an Intel Xeon X5650 at 2.67 GHz running Linux 64 bits, Gmp 5.0.2 [G+91] and setting $p = 16411$ a 15-bit prime number.

We start by giving some comparison of timings between the relaxed and zealous product in $(\mathbb{F}_p[[X]])[Y]$ depending on the degree in Y and the precision N of power series.

	Degree 32 in Y		Degree 64 in Y		Degree 128 in Y		Degree 256 in Y	
N	zealous	relaxed	zealous	relaxed	zealous	relaxed	zealous	relaxed
8	0	0	0	1	1	1	2	4
16	0	1	0	3	1	5	3	11
32	0	3	1	7	2	14	6	34
64	1	8	3	18	6	41	12	100
128	3	21	6	49	12	110	30	270
256	6	56	12	130	29	300	70	700
512	12	150	30	340	71	790	170	1800
1024	29	370	71	860	170	2000	340	4500
2048	72	920	170	2100	350	4800	750	11000

Table 6.1. Timings of zealous and relaxed multiplication in $(\mathbb{F}_p[[X]])[Y]$

We observe that the ratio of the timings between the relaxed and zealous algorithms grows as $\log(N)$. As a future work, we will apply the new relaxed multiplications of [Hoe07], generalized to more general ring in [Hoe12], to keep the ratio $\mathsf{R}(N)/\mathsf{I}(N)$ smaller.

To be fair, we mention that our zealous lifting implementation could also be improved. Especially the multiplication of matrices with polynomial entries could have benefit from an interpolation/evaluation scheme, especially since our matrices have reasonable size $n \times n$ and big entries in $(\mathbb{F}_p[[X]])[Y]$.

We tried our algorithm on two family of examples. The Katsura polynomials systems comes from a problem of magnetism in physics [Kat90]. The system Katsura-n has $n+1$ unknowns $X_0, \ldots, X_n$ and $n+1$ equations:

$$\text{for } 0 \leqslant m < n, \quad \sum_{\ell=-n}^{n} X_{|\ell|}\, X_{|m-\ell|} = X_m$$

and $X_0 + 2\sum_{\ell=1}^{n} X_\ell = 1$.

The other family of polynomial system MulLinForm-n has n unknowns and n equations of the form

$$(\lambda_1 X_1 + \cdots + \lambda_n X_n)\,(\mu_1 X_1 + \cdots + \mu_n X_n) = \alpha$$

where the λ_i, μ_i and α are random coefficients in $\mathbb{F}_p$.

We indicate with a bold font the theoretical bound for the precision of power series required in the KRONECKER algorithm.

	KATSURA-3		KATSURA-4		KATSURA-5		KATSURA-6	
N	zealous	relaxed	zealous	relaxed	zealous	relaxed	zealous	relaxed
2	21	7	75	20	250	58	780	170
4	31	11	106	29	350	78	1100	220
8	**49**	**18**	170	48	550	130	1700	360
16	82	36	**290**	**92**	940	240	1900	700
32	140	74	510	200	**1700**	**530**	5200	1500
64	260	160	970	440	3300	1200	**10000**	**3600**
128	510	360	1900	1000	6600	2800	21000	8600
256	1000	820	4000	2400				
512	2200	1900	8600	5500				

Table 6.2. Timings of zealous/relaxed lifting of univariate representations for KATSURA-n.

	MULLINFORM-4		MULLINFORM-5		MULLINFORM-6	
N	zealous	relaxed	zealous	relaxed	zealous	relaxed
2	44	16	160	45	520	130
4	64	23	230	63	720	180
8	96	38	340	100	1000	300
16	**150**	**69**	520	180	1700	540
32	230	140	**850**	**380**	2900	1000
64	370	180	1400	780	**5200**	**2300**
128	670	580	2600	1600	9500	4800

Table 6.3. Zealous/relaxed lifting timings of univariate representations of MULLINFORM-n.

6.5.2 Conclusion

As a conclusion, we remark that a relaxed approach has generated new algorithms for the lifting of triangular sets. Our hope was to save the cost of linear algebra in the dominant part of the complexity when the precision N tends to the infinity. Besides, previous experiences, with e.g. the relaxed lifting of regular roots, showed that we could expect to do less multiplications in the relaxed model than in the zealous one. Therefore, whenever the precision N gives a measured ratio of complexity between relaxed and zealous multiplication, we can expect better timings from the relaxed approach.

In view of our hopes, we are not completely satisfied with the lifting of general triangular sets, for it does more multiplications when $n\,L \geqslant n^\omega$. On the contrary, the lifting of univariate representations always improve the asymptotic number of multiplications.

Partie IV

A special algebraic system

Chapitre 7
Algorithms for the universal decomposition algebra

Let $\Bbbk$ be a field and let $f \in \Bbbk[T]$ be a polynomial of degree n. The *universal decomposition algebra* $\mathbb{A}$ is the quotient of $\Bbbk[X_1, ..., X_n]$ by the ideal of *symmetric relations* (those polynomials that vanish on all permutations of the roots of f). We show how to obtain efficient algorithms to compute in $\mathbb{A}$. We use a univariate representation of $\mathbb{A}$, *i.e.* an isomorphism of the form $\mathbb{A} \simeq \Bbbk[T]/Q(T)$, since in this representation, arithmetic operations in $\mathbb{A}$ are known to be quasi-optimal. We give details for two related algorithms, to find the isomorphism above, and to compute the characteristic polynomial of any element of $\mathbb{A}$.

7.1 Introduction

Let $\Bbbk$ be a field and let $f = X^n + \sum_{i=1}^{n} (-1)^i f_i X^{n-i}$ in $\Bbbk[X]$ be a degree n separable polynomial. We let $\mathcal{R} := \{\alpha_1, ..., \alpha_n\}$ be the set of roots of f in an algebraic closure of $\Bbbk$. The *ideal of symmetric relations* $\mathcal{I}_s$ is the ideal

$$\{P \in \Bbbk[X_1, ..., X_n] | \forall \sigma \in \mathfrak{S}_n, P(\alpha_{\sigma(1)}, ..., \alpha_{\sigma(n)}) = 0\}.$$

It is is generated by $(E_i - f_i)_{i=1,...,n}$, where E_i is the ith elementary symmetric function on $X_1, ..., X_n$. Finally, the *universal decomposition algebra* is the quotient algebra $\mathbb{A} := \Bbbk[X_1, ..., X_n]/\mathcal{I}_s$, of dimension $\delta := n!$. For all $P \in \mathbb{A}$, we denote by $\mathcal{X}_{P,\mathbb{A}}$ its characteristic polynomial in $\mathbb{A}$, that is, the characteristic polynomial of the multiplication-by-P endomorphism of $\mathbb{A}$. Stickelberger's theorem shows that

$$\mathcal{X}_{P,\mathbb{A}}(T) = \prod_{\sigma \in \mathfrak{S}_n} (T - P(\alpha_{\sigma(1)}, ..., \alpha_{\sigma(n)})) \in \Bbbk[T]. \tag{7.1}$$

This polynomial is related to the *absolute Lagrange resolvent*

$$L_P(T) := \prod_{\mathrm{Stab}(P)\backslash\backslash\mathfrak{S}_n} (T - P(\alpha_{\sigma(1)}, ..., \alpha_{\sigma(n)})) \in \Bbbk[T],$$

where $\mathrm{Stab}(P)\backslash\backslash\mathfrak{S}_n$ are the left cosets of the stabilizer of P in the symmetric group $\mathfrak{S}_n$; indeed, these polynomials satisfy the relation $\mathcal{X}_{P,\mathbb{A}} = L_P^{\#\mathrm{Stab}(P)}$.

Computing Lagrange resolvents is a fundamental question, motivated for instance by applications to Galois theory or effective invariant theory. There exists an abundant literature on this question [Lag70, Soi81, Val89, AV94, AV97, Leh97, Yok97, RV99, AV00]; known symbolic methods rely on techniques involving resultants, symmetric functions, standard bases or invariants (we will make use of some of these ingredients as well). However, little is known about the complexity of these methods. As it turns out, almost all algorithms have at least a quadratic cost δ^2 in the general case.

In some particular cases, though, it is known that resolvents can be computed in quasi-linear time [CM94]. Our goal in this article is thus to shed some light on these questions, from the complexity viewpoint: is it possible to give fast algorithms (as close to quasi-linear time as possible) for general P? What are some particular cases for which better solutions exist? To answer these questions, we measure the cost of our algorithms by the number of arithmetic operations in $\Bbbk$ they perform. Practically, this is well adapted to cases where $\Bbbk$ is a finite field; over $\Bbbk = \mathbb{Q}$, a combination of our results and modular techniques, such as in [Ren04] for resolvents, should be used.

The heart of the article, and the key to obtain better algorithms, is the question of which representation should be used for $\mathbb{A}$. A commonly used representation is *triangular*. The *divided differences*, also known as *Cauchy modules* [Che50, RV99], are defined by $C_1(X_1) := f(X_1)$ and

$$C_{i+1} := \frac{C_i(X_1, ..., X_i) - C_i(X_1, ..., X_{i-1}, X_{i+1})}{X_i - X_{i+1}} \tag{7.2}$$

for $1 \leqslant i < n$. They form a *triangular basis* of $\mathcal{I}_s$, in the sense that C_i is in $\Bbbk[X_1, ..., X_i]$, monic in X_i and reduced with respect to $(C_1, ..., C_{i-1})$. In particular, they define a tower of intermediate algebras $\mathbb{A}_i$ for $1 \leqslant i \leqslant n$:

$$\begin{gathered}
\mathbb{A}_1 := \Bbbk[X_1]/(C_1) \\
\vdots \\
\mathbb{A}_m := \Bbbk[X_1, ..., X_m]/(C_1, ..., C_m) \\
\vdots \\
\mathbb{A} = \mathbb{A}_n := \Bbbk[X_1, ..., X_n]/(C_1, ..., C_n).
\end{gathered}$$

In this approach, elements of $\mathbb{A}$ are represented by means of multivariate polynomials reduced modulo $(C_1, ..., C_n)$. For all $m \leqslant n$, $\mathbb{A}_m$ has dimension $\delta_m := n!/(n-m)!$; its elements are represented as polynomials in $X_1, ..., X_m$.

Introducing these intermediate algebras makes it possible for us to refine our problem: we will also consider the question of fast arithmetics, and in particular characteristic polynomial computation for $\mathbb{A}_m$. The characteristic polynomial of $P \in \mathbb{A}_m$ will be written $\mathcal{X}_{P,\mathbb{A}_m} \in \Bbbk[T]$; it has degree δ_m and admits the factorization

$$\mathcal{X}_{P,\mathbb{A}_m} = \prod_{\alpha_1,...,\alpha_m \in \mathcal{R}\,\text{pairwise}} (T - P(\alpha_1, ..., \alpha_m)). \tag{7.3}$$

Divided differences are inexpensive to compute via their recursive formula, but it is difficult to make computations in $\mathbb{A}_m$ efficient with this representation. To review known results, it will be helpful to consider two extreme cases: when m is small (typically, m is a constant), and when m is close to n. Note that the first case covers some useful cases for Galois theory (such as the computation of resolvents associated to simple polynomials of the form $X_1 X_2 + X_3 X_4$, ...).

When m is fixed (say $m = 4$ in the above example) and $n \to \infty$, $\delta_m = n!/(n-m)!$ is equivalent to n^m. In this case, there exist algorithms of cost $\tilde{\mathcal{O}}(\delta_m) = \tilde{\mathcal{O}}(n^m)$ for multiplication and inversion (when possible) in $\mathbb{A}_m$ [DMMSX06, LMS09]. Here, and everywhere else in this chapter, the $\tilde{\mathcal{O}}$ notation indicates the omission of logarithmic factors. For characteristic polynomial computation, it is possible to deduce from [LMP09] a cost estimate of $\tilde{\mathcal{O}}(\delta_m\, n^2) = \tilde{\mathcal{O}}(n^{m+2})$.

However, all these algorithms hide exponential factors in m in their big-O, which makes them unsuitable for the case $m \simeq n$. For the case $m = n$, the paper [BCHS11] gives a multiplication algorithm of cost $\tilde{\mathcal{O}}(\delta_n)$, but this algorithm hides high degree logarithmic terms in the big-O. There is no known quasi-linear algorithm for inverting elements of $\mathbb{A}_n$.

The second representation we discuss is univariate. For $m \leqslant n$, an element P of $\mathbb{A}_m$ will be called *primitive* if the $\Bbbk$-algebra $\Bbbk[P]$ spanned by P is equal to $\mathbb{A}_m$ itself. If Λ is a primitive linear form in $\mathbb{A}_m$, a *univariate representation* of $\mathbb{A}_m$ consists of polynomials $\mathfrak{P} = (Q, S_1, ..., S_m)$ in $\Bbbk[T]$ with $Q = \mathcal{X}_{\Lambda,\mathbb{A}_m}$ and $\deg(S_i) < \delta_m$ for all $i \leqslant m$ such that we have a $\Bbbk$-algebra isomorphism

$$\begin{array}{ccc} \mathbb{A}_m = \Bbbk[X_1, ..., X_m]/(C_1, ..., C_m) & \to & \Bbbk[T]/(Q) \\ X_1, ..., X_m & \mapsto & S_1, ..., S_m \\ \Lambda & \mapsto & T. \end{array}$$

A brief history of univariate representations and triangular sets can be found in Section 6.6.1.1.

When using univariate representations, the elements of $\mathbb{A}_m \simeq \Bbbk[T]/(Q)$ are then represented as univariate polynomials of degree less than δ_m. As usual, we will thus denote by $\mathsf{M}(n)$ the cost of polynomial multiplication for polynomials of degrees bounded by n, under the super-linearity assumptions of [GG03]. One can take $\mathsf{M}(n) = \mathcal{O}(n \log(n) \log(\log(n)))$ using Fast Fourier Transform [SS71, CK91].

Then, multiplications and inversions (when possible) in $\mathbb{A}_m$ cost respectively $\mathcal{O}(\mathsf{M}(\delta_m))$ and $\mathcal{O}(\mathsf{M}(\delta_m) \log(\delta_m))$. For characteristic polynomial, the situation is not as good, as no quasi-linear algorithm is known: the best known result [Sho94] is $\mathcal{O}(\mathsf{M}(\delta_m)\ \delta_m^{1/2} + \delta_m^{(\omega+1)/2})$. Here, ω is so that we can multiply $n \times n$ matrices within $\mathcal{O}(n^\omega)$ ring operations on any ring R. The best known bound on ω is $\omega \leqslant 2.3727$ [CW90, Sto10, VW11], resulting in a $\mathcal{O}(\delta_m^{1.69})$ characteristic polynomial algorithm.

Computing a univariate representation for $\mathbb{A}_m$ is expensive: for $m = n$, starting from any defining equations of $\mathcal{I}_s$, it takes time $\tilde{\mathcal{O}}(\delta_n^2)$ with the geometric resolution algorithm [GLS01]. Starting from the divided differences, the RUR algorithm [Rou99] or the FGLM algorithm [FGLM93] take time $\mathcal{O}(\delta_n^3)$; a recent improvement of the latter [FM11] could reduce the exponent using sparse linear algebra techniques. Some other algorithms are specifically designed to take as input a triangular set (such as the divided differences) and convert it to a univariate representation, such as [BLMM01] or [PS11]; the latter performs the conversion for any m in subquadratic time $\tilde{\mathcal{O}}(\mathsf{M}(\delta_m)\, \delta_m^{1/2} + \delta_m^{(\omega+1)/2})$, which is $\tilde{\mathcal{O}}(\delta_m^{1.69})$.

Thus, the triangular representation for $\mathbb{A}_m$ is easy to compute but leads to rather inefficient algorithms to compute in $\mathbb{A}_m$. On the other hand, computing a univariate representation is not straightforward, but once it is known, some computations in $\mathbb{A}_m$ become faster. Our main contribution in this chapter is to show how to circumvent the downsides of univariate representations, by providing fast algorithms for their construction (for $\mathbb{A}_n$ itself, or for each $\mathbb{A}_m$) in many cases. We also show how to use fast univariate arithmetics in $\mathbb{A}_m$ to compute characteristic polynomials efficiently.

We give two kinds of estimates, depending on whether m is fixed or not. In the first case, we are interested in what happens when $n \to \infty$; the big-O estimates may hide constants depending on m. In the second case, when both m and n can vary, a statement of the form $f(m,n) = \mathcal{O}(g(m,n))$ means that there exists K such that $f(m,n) \leqslant K\, g(m,n)$ holds for all m, n. For univariate representations, our algorithms are Las Vegas: we give *expected* running times.

Theorem 7.1. *Let $m \leqslant n$ and suppose that the characteristic of $\Bbbk$ is zero, or at least $2\,\delta_m^2$. Then we can compute characteristic polynomials and univariate representations in $\mathbb{A}_m$ with costs as specified in the following table.*

	$\mathcal{X}_{P,\mathbb{A}_m}$	*univ. representation (expected time)*
m fixed	$\mathcal{O}(\mathsf{M}(\delta_m))$ *for P linear*	$\mathcal{O}(\mathsf{M}(\delta_m)\log(n))$
$m \leqslant n/2$	$\mathcal{O}(n\,m\,\mathsf{M}(\delta_m))$ *for P linear*	$\mathcal{O}(n\,m^2\,\mathsf{M}(\delta_m))$
any m	$\mathcal{O}(n^{(\omega+1)/2}\,m\,\mathsf{M}(\delta_m))$	$\mathcal{O}(n^{(\omega+1)/2}\,m\,\mathsf{M}(\delta_m))$

In particular, when m is fixed, we have optimal algorithms (up to logarithmic factors) for characteristic polynomials of linear forms. For arbitrary P, the results in the last item are not optimal: when m is fixed, the running time of our algorithm is $\tilde{\mathcal{O}}(n^{m+1.69})$, for an output of size n^m. For small values of m, say $m = 2, 3, 4$, this is a significant overhead. However, these results do improve on the state-of-the-art.

We propose two approaches; both of them rely on classical ideas. The first one (in Section 7.3) computes characteristic polynomials by means of their Newton sums, following previous work of [Val89, AV94, CM94], but is limited to simple polynomials, such as linear forms; this will establish the first two items in the theorem. The second one (in Section 7.4) relies on iterated resultants [Lag70, Soi81, Leh97, RV99] and provides the last statements in the theorem. The last section gives experimental results.

In addition to the general results given in the theorem above, the following sections also mention other examples for which our techniques, or slight extensions thereof, yield quasi-linear results – as of now, we do not have a complete classification of all examples for which this is the case.

In all this chapter, our focus is on computing characteristic polynomials rather than resolvents. From this, one can deduce resolvents by root extraction, but it is of course preferable to compute the resolvent directly, by cleaning multiplicities as early as possible. The basic ideas we use are known to make this possible: we mention it in the next section for the Newton sums approach and [Leh97, RV99, AV12] discuss the resultant-based approach. However, quantifying the complexity gains of this improvement is beyond the scope of this chapter. Note also that for cases where P is fixed, such as $P := X_1 X_2 + X_3 X_4$, and $n \to \infty$, we can save only a constant factor in the running time with such considerations.

7.2 Preliminaries

7.2.1 The Newton representation

Let g be monic of degree n in $\Bbbk[X]$, and let β_1, ..., β_n its roots in an algebraic closure of $\Bbbk$. For $i \in \mathbb{N}$, we let $S_i(g) \in \Bbbk$ be the ith *Newton sum* of g, defined by $S_i(g) := \sum_{\ell=1}^{n} \beta_\ell^i$, and for $m \in \mathbb{N}$ we write $S(g,m) := (S_i(g))_{0 \leqslant i \leqslant m}$.

The conversion from coefficients to the Newton representation $S(g,m)$ and back can be done by the Newton-Girard formulas, but this takes quadratic time in m. To achieve a quasi-linear complexity, we recall a result first due to Schönhage [Sch82]; see [Bos03] for references and a more detailed exposition, including the proofs of the results we state below.

Lemma 7.2. *Let g be a monic polynomial of degree n in $\Bbbk[X]$. Then, for $m \in \mathbb{N}$, one can compute $S(g,m)$ in time $\mathcal{O}(\mathsf{M}(m))$. If the characteristic of $\Bbbk$ is either zero or greater than n, one can recover g from $S(g,n)$ in time $\mathcal{O}(\mathsf{M}(n))$.*

In particular, knowing $S(g,n)$, we can compute $S(g,n')$ for any $n' \geqslant n$ in time $\mathcal{O}(\mathsf{M}(n'))$.

The Newton representation is useful to speed up certain polynomial operations, such as multiplication and exact division, since $S_i(g\,h) = S_i(g) + S_i(h)$ for all $i \in \mathbb{N}$. Other improved operations include the *composed sum* and *composed product* of g and another polynomial h, with roots $\gamma_1, \ldots, \gamma_m$; they are defined by

$$\begin{aligned} g \oplus h &:= \prod_{i=1\ldots n, j=1\ldots m} (X - (\beta_i + \gamma_j)), \\ g \otimes h &:= \prod_{i=1\ldots n, j=1\ldots m} (X - (\beta_i \gamma_j)). \end{aligned}$$

Lemma 7.3. ([Bos03, section 7.3]) *Let g, h be monic polynomials in $\Bbbk[X]$, and suppose that $S(g,r)$ and $S(h,r)$ are known. Then one can compute $S(g \otimes h, r)$ in time $\mathcal{O}(r)$; if the characteristic of $\Bbbk$ is either zero or greater than r, one can compute $S(g \oplus h, r)$ in time $\mathcal{O}(\mathsf{M}(r))$.*

We write $\otimes_{\mathrm{NS}}(S(g,r), S(h,r), r)$ and $\oplus_{\mathrm{NS}}(S(g,r), S(h,r), r)$ for these algorithms; the subscript $_{\mathrm{NS}}$ shows that the inputs and outputs are in the Newton representation.

7.2.2 Univariate representations

We recall a few facts on univariate representations. Let us fix $m \leqslant n$. Then, a linear form Λ is primitive for $\mathbb{A}_m$ if and only if it takes distinct values on the points of the variety defined by $\mathcal{I}_s \cap \Bbbk[X_1, ..., X_m]$. This is the case if and only if the minimal polynomial of Λ coincides with its characteristic polynomial $\mathcal{X}_{\Lambda,\mathbb{A}_m}$, if and only if $\mathcal{X}_{\Lambda,\mathbb{A}_m}$ is squarefree. For instance when $m = n$, Λ is primitive in $\mathbb{A}_n$ if and only if the values $\Lambda(\alpha_{\sigma(1)}, ..., \alpha_{\sigma(n)})$ are all distinct for $\sigma \in \mathfrak{S}_n$.

By Zippel-Schwartz lemma [Zip79, Sch80], for $K \in \mathbb{N}_{>0}$, a random linear form Λ will be primitive for $\mathbb{A}_m$ with probability greater than $1 - 1/(2K)$ if its coefficients are taken in a set of cardinality $K\delta_m^2$; this still holds if we set $\lambda_1 := 1$. One can find primitive linear forms for $\mathbb{A}_m$ in a (non-uniform) deterministic manner, but with a cost polynomial in δ_m [CG10].

When Λ is primitive, in the univariate representation $\mathfrak{P} = (Q,\ S_1,\ ...,\ S_n)$ corresponding to Λ, we obtain Q as $Q = \mathcal{X}_{\Lambda,\mathbb{A}_m}$. The polynomials S_i are called *parametrizations* because they are the images of the variables X_i by the isomorphism $\mathbb{A}_m \simeq \Bbbk[T]/(Q)$. We will now argue that any "reasonable" algorithm that computes Q will also give us the parametrizations for a moderate overhead.

Let us extend the base field $\Bbbk$ to $\Bbbk' := \Bbbk(L_1, ..., L_m)$, where L_i are new indeterminates. Let $\mathbb{A}'_m := \mathbb{A}_m \otimes_\Bbbk \Bbbk'$ be obtained by adding $L_1, ..., L_m$ to the ground field in $\mathbb{A}_m$, and let finally $\mathcal{X}_{L,\mathbb{A}'_m} \in \Bbbk'[T]$ be the characteristic polynomial of $L := L_1 X_1 + \cdots + L_n X_m$. Then, the following holds:

$$S_i = -\frac{\partial \mathcal{X}_{L,\mathbb{A}'_m}}{\partial L_i} \Big/ \frac{\partial \mathcal{X}_{L,\mathbb{A}'_m}}{\partial T} \bmod \mathcal{X}_{L,\mathbb{A}'_m} \Big|_{L_1,...,L_m = \lambda_1,...,\lambda_m};$$

see for instance [Kro82, Kön03, Mac16, HKP+00, GLS01, DL08].

We can avoid working with m-variate rational function coefficients, as the formula above implies that we can obtain S_i as follows. Let $\Bbbk_\varepsilon := \Bbbk[\varepsilon]/(\varepsilon^2)$. For a given Λ, and for $i \leqslant m$, let $\mathcal{X}_{\Lambda_i}$ be the characteristic polynomial of $\Lambda_i := \Lambda + \varepsilon X_i$, computed over $\Bbbk_\varepsilon$. Then, $\mathcal{X}_{\Lambda_i}$ takes the form $\mathcal{X}_{\Lambda_i} = \mathcal{X}_{\Lambda,\mathbb{A}_m} + \varepsilon R_i$, and we obtain S_i as $S_i = R_i / \mathcal{X}'_{\Lambda,\mathbb{A}_m} \bmod \mathcal{X}_{\Lambda,\mathbb{A}_m}$.

We will require that the algorithm computing $\mathcal{X}_{\Lambda,\mathbb{A}_m}$ performs no zero-test or division (other than by constants in $\Bbbk$, since those can be seen as multiplications by constants). Since any ring operation $(+, \times)$ in $\Bbbk_\varepsilon$ costs at most 3 operations in $\Bbbk$, given such an algorithm that computes the characteristic polynomial of any linear form in $\mathbb{A}_m$ in time $\mathcal{C}$, we can deduce an algorithm that computes each S_i in time $\mathcal{O}(\mathcal{C})$, and $S_1, ..., S_m$ in time $\mathcal{O}(m\mathcal{C})$.

7.3 Newton sums techniques

In this section, we give our first algorithm for computing characteristic polynomials in $\mathbb{A}_m$. This approach is based on the following proposition and as such applies only to polynomials satisfying certain assumptions; the main result in this section is in Proposition 7.5 below. Our approach relies on Newton sums computations, following [Lag70, Val89, AV94, CM94]; an analogue of the following result can be found in [CM94] for the special cases $P = X_1 + \cdots + X_m$ and $P = X_1 \cdots X_m$. See also [BFSS06] for similar considerations in the bivariate case.

Proposition 7.4. *Let $P \in \mathbb{A}_m$ be of the form*

$$P(X_1, ..., X_m) := Q(X_1, ..., X_{m-1}) + R(X_m),$$

with Q in $\mathbb{A}_{m-1}$. For $1 \leqslant i \leqslant m-1$, define

$$P_i := Q(X_1, ..., X_{m-1}) + R(X_i) \in \mathbb{A}_{m-1},$$

and let $R_1 := R(X_1) \in \mathbb{A}_1$. Then the following equality holds:

$$\mathcal{X}_{P,\mathbb{A}_m} = \frac{\mathcal{X}_{Q,\mathbb{A}_{m-1}} \oplus \mathcal{X}_{R_1,\mathbb{A}_1}}{\prod_{i=1}^{m-1} \mathcal{X}_{P_i,\mathbb{A}_{m-1}}}. \tag{7.4}$$

Proof. Let $\mathcal{R} = \{\alpha_1, ..., \alpha_n\}$ be the roots of f and note that $\mathcal{X}_{R_1,\mathbb{A}_1} = \prod_{i=1}^{n} (T - R(\alpha_i))$. We rewrite (7.3) as

$$\mathcal{X}_{Q,\mathbb{A}_{m-1}} = \prod_{\alpha_1,...,\alpha_{m-1} \in \mathcal{R}\text{pairwise}} (T - Q(\alpha_1, ..., \alpha_{m-1})).$$

Thus, $\mathcal{X}_{Q,\mathbb{A}_{m-1}} \oplus \mathcal{X}_{R_1,\mathbb{A}_1}$ equals

$$\prod_{\alpha_1,...,\alpha_{m-1} \in \mathcal{R}\text{pairwise}, \alpha_m \in \mathcal{R}} (T - P(\alpha_1, ..., \alpha_m)).$$

This product contains parasite factors compared to $\mathcal{X}_{P,\mathbb{A}_m}$, corresponding to cases where $\alpha_m = \alpha_i$ for some i between 1 and $m-1$. For a given i, the factor due to $\alpha_m = \alpha_i$ is

$$\prod_{\alpha_1,...,\alpha_{m-1} \in \mathcal{R}\text{pairwise}} (T - P(\alpha_1, ..., \alpha_{m-1}, \alpha_i)),$$

that is, $\mathcal{X}_{P_i,\mathbb{A}_{m-1}}$. Formula (7.4) follows. □

This result can lead to a recursive algorithm, provided all recursive calls are well-defined (not all polynomials P satisfy the assumptions of this proposition). We will consider a convenient particular case, when the input polynomial is linear. In this case, we can continue the recursion all the way down, remarking that for $m = 1$, the characteristic polynomial of λX_1 is $f(\lambda T)$. We deduce our recursive algorithm CharNSRec, together with the top-level function CharNS; they compute $\mathcal{X}_{\Lambda,\mathbb{A}_m}$, for $\Lambda = \lambda_1 X_1 + \cdots + \lambda_m X_m$.

The algorithm CharNSRec uses the Newton sums representation for all polynomials involved; the only conversions are done in the top-level function CharNS. The algorithm thus takes as an extra argument the precision ℓ, that is, the number of Newton sums we need. As in the previous proposition, we write $\Lambda_0 := \lambda_1 X_1 + \cdots + \lambda_{m-1} X_{m-1}$ and, for $i \leqslant m$, $\Lambda_i := \lambda_1 X_1 + \cdots + \lambda_{m-1} X_{m-1} + \lambda_m X_i$.

Algorithm CharNSRec

Input: $S(f,n)$, m, Λ, the precision ℓ.
Output: $S(\mathcal{X}_{\Lambda,\mathbb{A}_m},\ell)$.

1. $\ell' := \min(\ell,\delta_m)$
2. **if** $(m=1)$ **then** out $:= (S_i(f)\,\lambda_1^i)_{0\leqslant i\leqslant n}$ **else**
   a. out $:=$ CharNSRec$(S(f,n),m-1,\Lambda_0,\ell')$
   b. out $:= \oplus_{\mathrm{NS}}$(out, CharNSRec$(S(f,n),1,\lambda_m X_1,\ell'),\ell')$
   c. **for** i **from** 1 **to** $m-1$
      out $:=$ out $-$ CharNSRec$(S(f,n),m-1,\Lambda_i,\ell')$
3. **if** $(\ell'<\ell)$ **then** Extend the series "out" up to precision ℓ
4. **return** out

The minus in step 2.c corresponds to a division in Formula (7.4). The main algorithm follows; it uses a trick when $m=n$ to reduce the depth of the recursion by one unit.

Algorithm CharNS

Input: f, m, Λ.
Output: $\mathcal{X}_{\Lambda,\mathbb{A}_m}$.

1. **if** $(m=n)$ **then**
   a. $\bar{\Lambda} := (\lambda_1-\lambda_n)X_1+\cdots+(\lambda_{n-1}-\lambda_n)X_{n-1}$
   b. **return** CharNS$(f,n-1,\bar{\Lambda})\oplus(X-\lambda_n f_1)$
2. Compute the Newton representation $S(f,n)$
3. $S(\mathcal{X}_{\Lambda,\mathbb{A}_m},\delta_m) :=$ CharNSRec$(S(f,n),m,\Lambda,\delta_m)$
4. Recover $\mathcal{X}_{\Lambda,\mathbb{A}_m}$ from $S(\mathcal{X}_{\Lambda,\mathbb{A}_m},\delta_m)$
5. **return** $\mathcal{X}_{\Lambda,\mathbb{A}_m}$

Proposition 7.5. *Let $m\leqslant n$ and suppose that the characteristic $\Bbbk$ is either zero or greater than δ_m. Then Algorithm* CharNS *computes the characteristic polynomials of linear forms in $\mathbb{A}_m$ in time $\mathcal{O}(\mathsf{M}(\delta_m))$ if m is bounded, $\mathcal{O}(m\,n\mathsf{M}(\delta_m))$ if $m\leqslant n/2$ and $\mathcal{O}(2^n\,\mathsf{M}(\delta_m))$ in general.*

Proof. Let be $\mathcal{C}(m,\ell)$ be the cost of CharNSRec on input $\Lambda\in\mathbb{A}_m$ and precision ℓ. We use the abbreviation $\mathcal{C}(m) := \mathcal{C}(m,\delta_m)$, so that $\mathcal{C}(1)=\mathcal{O}(n)$. For $2\leqslant m\leqslant n-1$, Lemma 7.2 gives $\mathcal{C}(m,\ell)=\mathcal{C}(m)+\mathcal{O}(\mathsf{M}(\ell))$ for $\ell\geqslant\delta_m$, so we get

$$\begin{aligned}\mathcal{C}(m) &= m\,\mathcal{C}(m-1,\delta_m)+\mathcal{C}(1,\delta_m)+\mathcal{O}(m\,\mathsf{M}(\delta_m))\\ &= m\,(\mathcal{C}(m-1)+\mathcal{O}(\mathsf{M}(\delta_m)))+\mathcal{O}(m\,\mathsf{M}(\delta_m))\\ &\leqslant m\,\mathcal{C}(m-1)+\mathcal{O}(m\,\mathsf{M}(\delta_m)).\end{aligned}$$

Then, by unrolling the recurrence and using the super-linearity of the function M, we deduce

$$\begin{aligned}\frac{\mathcal{C}(m)}{\mathsf{M}(\delta_m)} &\leqslant \mathcal{O}\left(m + m\,(m-1)\,\frac{\delta_{m-1}}{\delta_m} + \cdots + m!\,\frac{\delta_1}{\delta_m}\right)\\ &\leqslant \mathcal{O}\left(\sum_{i=1}^{m} \frac{m!}{(i-1)!}\,\frac{(n-m)!}{(n-i)!}\right)\\ &\leqslant \mathcal{O}\left(\frac{n}{\binom{n}{m}}\sum_{i=1}^{m}\binom{n-1}{i-1}\right).\end{aligned}$$

When m is bounded, the sum is bounded. If $m \leqslant n/2$, we derive the bound $\mathcal{C}(m) = \mathcal{O}(m\,n\,\mathsf{M}(\delta_m))$ from the remark $\binom{n-1}{i-1} \leqslant \binom{n}{i} \leqslant \binom{n}{m}$ for $1 \leqslant i \leqslant m$. For $m \leqslant n-1$, we get the cruder bound $\mathcal{C}(m) = \mathcal{O}(2^n\,\mathsf{M}(\delta_m))$. In all cases, the cost of Algorithm CharNS is the same, up to $\mathcal{O}(\mathsf{M}(\delta_m))$ for conversions. For $m = n$, let $\bar{\Lambda} := (\lambda_1 - \lambda_n)\,X_1 + \cdots + (\lambda_{n-1} - \lambda_n)\,X_{n-1}$. Then, $f_1 = \sum_i \alpha_i$ implies $\mathcal{X}_{\Lambda,\mathbb{A}_n} = \mathcal{X}_{\bar{\Lambda},\mathbb{A}_{n-1}} \oplus (X - \lambda_n\,f_1)$; the cost for $m = n$ is thus the same as for $m = n-1$, up to $\mathcal{O}(\mathsf{M}(\delta_n))$ for the composed sum. □

This proves the left-hand columns of the first two rows in Theorem 7.1. Using the discussion in Subsection 7.2.2, we can also compute a univariate representation of $\mathbb{A}_m$. After computing $\mathcal{X}_{\Lambda,\mathbb{A}_m}$, we test whether Λ is primitive for $\mathbb{A}_m$, by testing whether $\mathcal{X}_{\Lambda,\mathbb{A}_m}$ is squarefree; this takes time $\mathcal{O}(\mathsf{M}(\delta_m)\,\log\,(\delta_m))$, which is $\mathcal{O}(m\,\mathsf{M}(\delta_m)\log(n))$. If the characteristic of $\Bbbk$ is either zero, or at least equal to $2\,\delta_m^2$, we expect to try finitely many Λ before finding a primitive one. When this is the case, we can apply the procedure of Subsection 7.2.2 to obtain all parametrizations; this costs m times as much as computing $\mathcal{X}_{\Lambda,\mathbb{A}_m}$. Considering the cases m constant and $m \leqslant n/2$, this completes the proof of the first two points in our main theorem.

To conclude this section, we mention (without proof) some extensions. First, it is possible to adapt this algorithm to exploit symmetries of P, since they are known to create multiplicities in $\mathcal{X}_{P,\mathbb{A}_m}$: we can accordingly reduce the number of Newton sums we need (thus, one can compute resolvents directly in this manner). This is useful in practice, but we were not able to quantify the gains in terms of complexity.

Another remark is that an analogue to Proposition 7.4 holds for $P(X_1, \ldots, X_m) := Q(X_1, \ldots, X_{m-1}) \times R(X_m)$, replacing the operation $\oplus$ by $\otimes$. As an application, consider the case $P := X_1\,X_2\,X_3 + X_4$, so that $Q := X_1\,X_2\,X_3$ and $R := X_4$. To compute $\mathcal{X}_{P,\mathbb{A}_4}$, we are led to deal with Q, $P_1 := (1 + X_2\,X_3)\,X_1$, $P_2 := (1 + X_1\,X_3)\,X_2$ and $P_3 := (1 + X_1\,X_2)\,X_3$ in $\mathbb{A}_3$. By symmetry, it is enough to consider Q and P_3. For Q, we can continue the recursion all the way down to univariate polynomials, using the multiplicative version of the previous proposition. For P_3, however, we cannot. Writing P_3 as $(1 + X_1\,X_2) \times X_3$, the recursive call lead us in particular to compute the characteristic polynomial of $(1 + X_1\,X_2) \times X_2$, which does not satisfy the assumptions of the proposition.

Similar (but slightly more complicated) results hold when $P(X_1, \ldots, X_m)$ can be written as $Q(X_1, \ldots, X_\ell)\,\mathrm{op}\,R(X_{\ell+1}, \ldots, X_m)$, with $\mathrm{op} \in \{+, \times\}$. Taking for instance $P := X_1\,X_2 + X_3\,X_4$, we are led recursively to compute the characteristic polynomials of $X_1\,X_2$ and $P_1 := X_1\,(X_2 + X_3)$. However, the case of P_1 reduces to that of $X_2\,(X_2 + X_3)$, which does not satisfy the assumptions of the proposition. We will discuss these examples again in the next section.

7.4 Resultant techniques

Resultant methods to compute characteristic polynomials in $\mathbb{A}_m$ go back to Lagrange's elimination method (similar to today's resultant) to compute resolvents [Lag70]. This idea was developed in [Soi81, Leh97, RV99].

The basic idea is simple. Let again $C_1, ..., C_n$ be the divided differences associated to f. For $P \in \Bbbk[X_1, ..., X_m]$, define recursively the resultants

$$\begin{aligned} G_m &:= T - P(X_1, ..., X_m) \in \Bbbk[X_1, ..., X_m, T], \\ G_i &:= \mathrm{Res}_{X_{i+1}}(C_{i+1}, G_{i+1}) \in \Bbbk[X_1, ..., X_i, T], \end{aligned}$$

for $i = m-1, ..., 0$, so that $\mathcal{X}_{P,\mathbb{A}_m} = G_0 \in \Bbbk[T]$. In order to avoid an exponential growth of the degrees in the intermediate G_i's, we need to compute the resultant $\mathrm{Res}_{X_i}(C_i, G_i)$ over the coefficient ring $\mathbb{A}_{i-1}[T]$.

However, we mentioned that arithmetic in $\mathbb{A}_{i-1}$ is rather slow; univariate computations are faster. We give below a general framework that relies on both triangular and univariate representations to compute efficiently such resultants. Recall that a family of polynomials $\boldsymbol{T} = (T_1, ..., T_m)$ in $\Bbbk[X_1, ..., X_m]$ is a *triangular set* if the following holds for all $i \leqslant m$: T_i is in $\Bbbk[X_1, ..., X_i]$, T_i is monic in X_i and T_i is reduced with respect to $(T_1, ..., T_{i-1})$. Our main idea holds for general triangular families of polynomials, but it is only for the special case of divided difference that it will lead to an efficient algorithm (see Corollary 7.11 below).

7.4.1 General algorithms

In this section, we describe a general approach to compute characteristic polynomials modulo a triangular set. Following [DFS09, PS11], our main idea is to introduce *mixed* representations, that allow one to convert from triangular to *bivariate* representations, and back, one variable at a time.

Let $\boldsymbol{T} = (T_1, ..., T_m)$ be a triangular set in $\Bbbk[X_1, ..., X_m]$. For $i \leqslant m$, let $d_i := \deg(T_i, X_i)$, $\mu_i := d_1 \cdots d_i$ and $\mu'_i := d_{i+1} \cdots d_m$. We write $R_{\boldsymbol{T}} := \Bbbk[X_1, ..., X_m]/(T_1, ..., T_m)$; this is a $\Bbbk$-algebra of dimension $\mu_m = d_1 \cdots d_m$. More generally, for $i \leqslant m$, we write $R_{\boldsymbol{T},i} := \Bbbk[X_1, ..., X_i]/(T_1, ..., T_i)$; this is a $\Bbbk$-algebra of dimension μ_i.

Generalizing the notation used up to now, for P in $R_{\boldsymbol{T}}$, we write $\mathcal{X}_{P,R_{\boldsymbol{T}}}$ for its characteristic polynomial in $R_{\boldsymbol{T}}$, that is, the characteristic polynomial of the multiplication-by-P endomorphism of $R_{\boldsymbol{T}}$. To compute $\mathcal{X}_{P,R_{\boldsymbol{T}}}$, we will use the "iterated resultant" techniques sketched in the preamble.

Since computing modulo triangular sets is difficult, our workaround is to introduce a family of univariate representations $\mathfrak{P}_1$, ..., $\mathfrak{P}_{m-1}$ of respectively $R_{\boldsymbol{T},1}$, ..., $R_{\boldsymbol{T},m-1}$; in the introduction, we only defined univariate representations for the algebras $\mathbb{A}_i$, but the definition carries over unchanged to this slightly more general context [GLS01, PS11]. For $i \leqslant m-1$, $\mathfrak{P}_i$ has the form $\mathfrak{P}_i = (Q_i, S_{i,1}, ..., S_{i,i})$, with all polynomials in $\Bbbk[Z_i]$ and with associated linear form $\Lambda_i := \lambda_{i,1} X_1 + \cdots + \lambda_{i,i} X_i$. For $i = 1$, we add *w.l.o.g.* the mild restriction that $\Lambda_1 = X_1$, so that $Q_1 = T_1$.

We first show how to use these objects to perform conversions between multivariate and bivariate representations, going one variable at a time. For $i \leqslant m-1$, we know that Q_i has degree μ_i and that we have the $\Bbbk$-algebra isomorphism

$$\begin{array}{rccc} & R_{\boldsymbol{T},i} & \longrightarrow & \Bbbk[Z_i]/(Q_i) \\ \varphi_i\colon & X_1, \ldots, X_i & \longmapsto & S_{1,i}, \ldots, S_{i,i} \\ & \Lambda_i & \longmapsto & Z_i. \end{array}$$

We extend φ_i to another isomorphism

$$\Phi_i\colon \quad R_{\boldsymbol{T},i}[X_{i+1}, \ldots, X_m] \longrightarrow \Bbbk[Z_i]/(Q_i)[X_{i+1}, \ldots, X_m],$$

where φ_i acts coefficientwise, and we define $Q_{i,j} = \Phi_i(T_j)$ for $i+1 \leqslant j \leqslant m$.

Let us see $Q_{i,i+1}, \ldots, Q_{i,m}$ in $\Bbbk[Z_i, X_{i+1}, \ldots, X_m]$, by taking their canonical preimages. Then, $(Q_i, Q_{i,i+1}, \ldots, Q_{i,m})$ form a triangular set in $\Bbbk[Z_i, X_{i+1}, \ldots, X_m]$, such that $\deg(Q_{i,j}, X_j) = \deg(T_j, X_j)$ for $i+1 \leqslant j \leqslant m$. For $i \leqslant m-1$ and $i \leqslant j \leqslant m$, we will write

$$R_{i,j} = \Bbbk[Z_i, X_{i+1}, \ldots, X_j]/(Q_i, Q_{i,i+1}, \ldots, Q_{i,j}).$$

Then, still acting coefficientwise in $X_{i+1}, \ldots, X_j$, φ_i extends to an isomorphism $\Phi_{i,j}\colon R_{\boldsymbol{T},j} \to R_{i,j}$.

Two operations will be needed to convert between the various induced representations: *lift-up* and *push-down* [DFS09, PS11]. For $i \leqslant m-2$ and $i+1 \leqslant j \leqslant m$, we call *lift-up* the change of basis $\mathsf{up}_{i,j} := \Phi_{i+1,j} \circ \Phi_{i,j}^{-1}$. This is thus an isomorphism $R_{i,j} \to R_{i+1,j}$, with

$$\begin{aligned} R_{i,j} &= \Bbbk[Z_i, X_{i+1}, \ldots, X_j]/(Q_i, Q_{i,i+1}, \ldots, Q_{i,j}), \\ R_{i+1,j} &= \Bbbk[Z_{i+1}, X_{i+2}, \ldots, X_j]/(Q_{i+1}, Q_{i+1,i+2}, \ldots, Q_{i+1,j}). \end{aligned}$$

In particular, with $j = i+1$, we write up_i instead of $\mathsf{up}_{i,i+1}$; thus, it is the bivariate-to-univariate conversion given by

$$\mathsf{up}_i\colon \quad \begin{array}{c} R_{i,i+1} = \Bbbk[Z_i, X_{i+1}]/(Q_i, Q_{i,i+1}) \\ \downarrow \\ R_{i+1,i+1} = \Bbbk[Z_{i+1}]/(Q_{i+1}). \end{array}$$

Conversely, we call *push-down* the inverse change of basis; as above, for $j = i+1$, we write $\mathsf{down}_i = \mathsf{down}_{i,i+1}$. The operations up_i and down_i are crucial, since all $\mathsf{up}_{i,j}$ (resp. $\mathsf{down}_{i,j}$), for $j \geqslant i+2$, are obtained by applying up_i (resp. down_i) coefficientwise. We do not discuss here how to implement them in general (see [PS11]); we will give a better solution in the case of divided differences below. For the moment, we simply record the following straightforward result.

Lemma 7.6. *For $i \leqslant m-2$, suppose that one can apply up_i (resp. down_i) using u_i (resp. v_i) operations in $\Bbbk$. Then, one can apply $\mathsf{up}_{i,m}$ using $u_i\,\mu'_{i+1}$ operations in $\Bbbk$ (resp. one can apply $\mathsf{down}_{i,m}$ using $v_i\,\mu'_{i+1}$ operations in $\Bbbk$).*

Finally, we define $\mathsf{Up}_m = \mathsf{up}_{m-2,m} \circ \cdots \circ \mathsf{up}_{1,m}$ and $\mathsf{Down}_m = \mathsf{Up}_m^{-1}$ so that we have

$$\begin{array}{c} R_{m-1,m} = \Bbbk[Z_{m-1}, Z_m]/(Q_{m-1}, Q_{m-1,m}) \\ {\scriptstyle \mathrm{Down_m}}\downarrow \qquad\qquad\qquad \uparrow{\scriptstyle \mathrm{Up_m}} \\ R_{\boldsymbol{T}} = \Bbbk[X_1, \ldots, X_m]/(T_1, \ldots, T_m). \end{array}$$

We could want to go all the way down to univariate polynomials instead of bivariate, but it would not be useful: the algorithm below uses bivariate polynomials. In terms of complexity, the following is a direct consequence of Lemma 7.6.

Lemma 7.7. *For $i \leqslant m-2$, suppose that one can apply up_i (resp. down_i) using u_i (resp. v_i) operations in $\Bbbk$. Then one can apply Up_m (resp. Down_m) in respective times*

$$\sum_{i=1}^{m-2} u_i \, \mu'_{i+1} \quad \text{and} \quad \sum_{i=1}^{m-2} v_i \, \mu'_{i+1}.$$

Now we explain how to compute $G := \mathcal{X}_{P,R_{\boldsymbol{T}}} \in \Bbbk[Y]$ for any P in $R_{\boldsymbol{T}}$. Let $\Bbbk' := \Bbbk[Y]$; then, $\boldsymbol{T}$ is also a triangular set in $\Bbbk'[X_1, \ldots, X_m]$, and we define, for $i \leqslant m$,

$$R'_{\boldsymbol{T},i} := \Bbbk'[X_1, \ldots, X_i]/(T_1, \ldots, T_i) = R_{\boldsymbol{T},i}[Y].$$

As explained in the preamble of this section, we start by defining $G_m := Y - P \in R'_{\boldsymbol{T},m}$. For $i = m-1, \ldots, 0$, suppose that we know $G_{i+1} \in R'_{\boldsymbol{T},i+1}$. Seeing $R'_{\boldsymbol{T},i+1}$ as $R'_{\boldsymbol{T},i+1} = R'_{\boldsymbol{T},i}[X_{i+1}]/(T_{i+1})$, we define

$$G_i := \mathrm{Res}_{X_{i+1}}(T_{i+1}, G_{i+1}) \in R'_{\boldsymbol{T},i}.$$

Standard properties of resultants (see e.g. [Bou73, § 12.2]) show that $G_0 = G$. By induction, we prove that $\deg(G_i, Y) = \mu'_i$; in particular, $\deg(G_0, Y) = \mu$, as it should be.

We are going to compute $G_{m-1}, \ldots, G_0$ assuming that we know the univariate representations $\mathfrak{P}_1, \ldots, \mathfrak{P}_{m-1}$, and use univariate arithmetic as much as possible. For $1 \leqslant i \leqslant m-1$ and $i \leqslant j \leqslant m$, $R'_{i,j}$ is well defined and isomorphic to $R'_{\boldsymbol{T},j}$ because $R'_{i,j} = R_{i,j}[Y]$ and $R'_{\boldsymbol{T},j} = R_{\boldsymbol{T},j}[Y]$. Besides, lift-up and push-down are still defined; they are written respectively $\mathsf{up}'_i \colon R'_{i,i+1} \to R'_{i+1,i+1}$ and down'_i.

Lemma 7.8. *For $i \leqslant m-2$, suppose that one can apply up_i (resp. down_i) using u_i (resp. v_i) operations in $\Bbbk$. Then, for F in $R'_{i,i+1}$, with $d := \deg(F, Y)$, we can compute $\mathsf{up}'_i(F) \in R'_{i+1,i+1}$ using $\mathcal{O}(d\, u_i)$ operations in $\Bbbk$. For F in $R'_{i+1,i+1}$, with $d := \deg(F, Y)$, we can compute $\mathsf{down}'_i(F) \in R'_{i,i+1}$ using $\mathcal{O}(d\, v_i)$ operations in $\Bbbk$.*

This leads to our algorithm for characteristic polynomials. For convenience, we let $R_{0,1} := R_1$, and we let down'_0 be the identity map. For the moment, we assume that all polynomials $Q_{i,i+1}$ needed below are already known.

Algorithm CharResultant

Input: P in $R_{\boldsymbol{T}}$.
Output: $\mathcal{X}_{P,R_{\boldsymbol{T}}}$.

1. $P' := \mathsf{Up}_m(P)$ $\quad P' \in R_{m-1,m}$
2. $G_m := Y - P'$ $\quad G'_m \in R'_{m-1,m}$
3. **for** $i = m-1, ..., 1$ **do**
   a. $G'_i := \mathrm{Res}_{X_{i+1}}(Q_{i,i+1}, G_{i+1}) G'_i \in R'_{i,i}$
   b. $G_i := \mathsf{down}'_{i-1}(G'_i)$ $\quad G_i \in R'_{i-1,i}$
4. **return** $G_0 = \mathrm{Res}_{X_1}(G_1, Q_1)$. $\quad G_0 \in R'$

To analyze this algorithm, we remark that over any ring R, resultants of polynomials of degree d in $R[X]$ can be computed in $\mathcal{O}(d^{(\omega+1)/2})$ ring operations, provided one of these polynomials is monic, and 1, ..., d are units in R. Indeed, the resultant $\mathrm{Res}_X(A, B)$, with A monic of degree d and $\deg(B, X) < d$ is the constant term of the characteristic polynomial of B modulo A. This whole polynomial can be computed in time $\mathcal{O}(d^{(\omega+1)/2})$ by an algorithm of Shoup [Sho94] which performs no zero-test and only divisions by $1, ..., d$.

Proposition 7.9. *Suppose that one can apply* up_i *(resp.* down_i*) using* u_i *(resp.* v_i*) operations in* $\Bbbk$*, and that* $\Bbbk$ *has characteristic either zero, or at least* μ_m*. Then Algorithm* CharResultant *computes* $\mathcal{X}_{P,R_{\boldsymbol{T}}}$ *in time*

$$\mathcal{O}\left(\sum_{i=1}^{m-2} (u_i + v_i)\, \mu'_{i+1} + \sum_{i=0}^{m-1} d_{i+1}^{(\omega+1)/2}\, \mathsf{M}(\mu_m) \right).$$

Proof. We have seen that Step 1 takes time $\sum_{i=1}^{m-2} u_i \mu'_{i+1}$. For $i = m-1, ..., 1$, G'_i has degree μ'_i in Y, so Step 3.b takes time $v_{i-1}\, \mu'_i$ by Lemma 7.8.

In Step 3.a, we compute G_i by evaluation / interpolation in the variable Y, using evaluation points in geometric progression [BS05]; such points exist by assumption on the characteristic of $\Bbbk$. Both G_{i+1} and $Q_{i,i+1}$ have degree at most d_{i+1} in X_{i+1}, and $\deg(G'_i, Y) = \mu'_i$. Thus, the cost is $\mathcal{O}(d_{i+1}\, \mathsf{M}(\mu'_i))$ operations in $R_{i,i}$ for all evaluations / interpolations. Since the evaluation points are in $\Bbbk$, evaluation and interpolation are $\Bbbk$-linear operations, so each of them uses μ_i operations in $\Bbbk$.

The cost for all individual resultants is $\mathcal{O}(\mu'_i\, d_{i+1}^{(\omega+1)/2})$ ring operations in $R_{i,i}$, each of which takes $\mathcal{O}(\mathsf{M}(\mu_i))$ operations in $\Bbbk$. The conclusion follows using the inequalities $\mu_i\, \mathsf{M}(\mu'_i) \leqslant \mathsf{M}(\mu_m)$ and $\mathsf{M}(\mu_i)\, \mu'_i \leqslant \mathsf{M}(\mu_m)$. □

7.4.2 The case of divided differences

We now apply the former results to the triangular set of divided differences. Fix $m \in \mathbb{N}$ such that $m \leqslant n$, and take $\boldsymbol{T} = (C_1, ..., C_m)$ in $\Bbbk[X_1, ..., X_m]$. Note that $d_i := \deg(C_i, X_i)$ is equal to $n + 1 - i \leqslant n$, and that $R_{\boldsymbol{T},i}$ becomes $\mathbb{A}_i$ for $1 \leqslant i \leqslant m$. We also have $\mu_i = \delta_i$ and $\mu'_i = \delta_m / \delta_i$.

We are going to study lift-up and push-down for divided differences, with the objective to give estimates on the quantities u_i and v_i defined above. Thus, we start from univariate representations $\mathfrak{P}_1, ..., \mathfrak{P}_{m-1}$ for $\mathbb{A}_1, ..., \mathbb{A}_{m-1}$; for the moment, they are part of the input.

We impose a further restriction on $\mathfrak{P}_1, ..., \mathfrak{P}_{m-1}$, assuming that for all $i < m-1$, $\Lambda_{i+1} = \Lambda_i + \lambda_{i+1} X_{i+1}$ for some λ_{i+1} in $\Bbbk$. When this is the case, we call $\mathfrak{P}_1$, ..., $\mathfrak{P}_{m-1}$ *compatible*. Then, we have $\Lambda_i = X_1 + \lambda_2 X_2 + \cdots + \lambda_i X_i$, since by assumption $\Lambda_1 = X_1$. Thus, compatible univariate representations are associated to a $(m-2)$-uple $(\lambda_2, ..., \lambda_{m-1}) \in \Bbbk^{m-2}$, with the condition that every $X_1 + \lambda_2 X_2 + \cdots + \lambda_i X_i$ is a primitive element of $\mathbb{A}_i$ for all $i \leqslant m-1$. Under this condition, we now study the cost of lift-up and push-down. Indeed, in this case, we can deduce the explicit form of up_i:

$$\mathsf{up}_i\colon \begin{array}{ccc} \Bbbk[Z_i, X_{i+1}]/(Q_i, Q_{i,i+1}) & \longrightarrow & \Bbbk[Z_{i+1}]/(Q_{i+1}) \\ Z_i & \longmapsto & Z_{i+1} - \lambda_{i+1} S_{i+1,i+1} \\ X_{i+1} & \longmapsto & S_{i+1,i+1} \\ Z_i + \lambda_{i+1} X_{i+1} & \longmapsto & Z_{i+1}. \end{array}$$

The key for the following algorithms is then the remark that $f(X_{i+1}) = 0$ in $\mathbb{A}_{i+1}$; we will exploit the fact that the polynomial f is a small degree, *univariate* polynomial. To analyze its cost, we will use the following bounds: for $\ell \geqslant 1$, consider the sum $S(m, n, \ell) := \sum_{1 \leqslant i \leqslant m} i^\ell \, \mathsf{M}(\delta_i)$. Then we claim that the following holds:

$$S(m, n, \ell) \leqslant \exp(1)\, m^\ell \, \mathsf{M}(\delta_m) = \mathcal{O}(m^\ell \, \mathsf{M}(\delta_m)).$$

Indeed,the super-linearity of the function M implies

$$\frac{S(m, n, \ell)}{\mathsf{M}(\delta_m)} \leqslant \sum_{1 \leqslant i \leqslant m} i^\ell \frac{\delta_i}{\delta_m} \leqslant m^\ell \sum_{1 \leqslant i \leqslant m} \frac{\delta_i}{\delta_m} \leqslant \sum_{i \in \mathbb{N}} \frac{1}{n!}.$$

Proposition 7.10. *Suppose that $\mathfrak{P}_1, ..., \mathfrak{P}_{m-1}$ are known and compatible. If the characteristic of $\Bbbk$ is either zero or at least δ_{m-1}, then for $1 \leqslant i \leqslant m-2$, up_i and down_i can be computed in time $u_i = \mathcal{O}(\mathsf{M}(n)\, \mathsf{M}(\delta_{i+1}))$ and $v_i = \mathcal{O}(\mathsf{M}(n)\, \mathsf{M}(\delta_{i+1}))$.*

Proof. First, we study the following simplified problem: given $\lambda \in \Bbbk$, some polynomials $A \in \Bbbk[Z]$, $B \in \Bbbk[Z, X]$ monic in X, and W, S in $\Bbbk[Z]$, compute the mapping

$$\mathsf{up}\colon \begin{array}{ccc} \Bbbk[Z, X]/(A, B) & \longrightarrow & \Bbbk[Z]/(W) \\ Z & \longmapsto & Z - \lambda S \\ X & \longmapsto & S \\ Z + \lambda X & \longmapsto & Z, \end{array}$$

and its inverse down, assuming up is well-defined and invertible. We write $a := \deg(A)$ and $b := \deg(B, X)$, so that $\deg(W) = a\, b$. We also assume that $f(X) = 0$ in $\Bbbk[Z, X]/(A, B)$, for some monic polynomial $f \in \Bbbk[X]$ of degree $n \geqslant b$. Finally, the characteristic of $\Bbbk$ is supposed to be either 0 or at least $a\, b$. Then, we show that both directions take time $\mathcal{O}(\mathsf{M}(n)\, \mathsf{M}(a\, b))$.

Computing up. Given $H \in \Bbbk[Z, X]/(A, B)$, we first show how to compute $G := \mathsf{up}(H)$. Let $H^\star$ be the canonical preimage of H in $\Bbbk[Z, X]$, so that $G = H^\star(Z - \lambda S, S) \bmod W$. Then, we obtain G as follows:

1. Compute $H^\star(Z - \lambda X, X)$ modulo f using the shift algorithm of [ASU75] (which is possible under our assumption on the characteristic of $\Bbbk$) with coefficients in $\Bbbk[X]/(f)$
2. Evaluate previous result at $X = S$ using Horner scheme.

Step 1 takes time $\mathcal{O}(\mathsf{M}(n)\,\mathsf{M}(a))$; the next step uses n multiplications modulo W, for a total of $\mathcal{O}(n\,\mathsf{M}(a\,b))$.

Computing down. Conversely, for $G \in \Bbbk[Z]/(W)$, we show how to compute $H := \mathsf{down}(G)$. Let $G^\star$ be the canonical preimage of G in $\Bbbk[Z]$, so that $H = G(Z + \lambda X) \bmod (A, B)$. We obtain H as follows:

1. Compute $G(Z + \lambda X)$ modulo f, using again the shift algorithm of [ASU75] with coefficients in $\Bbbk[X]/(f)$.
2. Reduce previous result modulo (A, B).

Step 1 takes time $\mathcal{O}(\mathsf{M}(n)\,\mathsf{M}(a\,b))$, then the reduction takes time $\mathcal{O}(\mathsf{M}(n)\,\mathsf{M}(a\,b))$ by fast Euclidean division.

Conclusion. By the former discussion, given $A = Q_i$, $B = Q_{i,i+1}$ and $W = Q_{i+1}$, up_i and down_i can be computed in time $u_i = \mathcal{O}(\mathsf{M}(n)\,\mathsf{M}(\delta_{i+1}))$.

First, though, we have to compute $Q_{i,i+1}$. Supposing that $Q_{i-1,i}$ is known, we can compute $Q_{i,i+1}$ by adjusting Formula (7.2), writing

$$Q_{i,i+1} = \mathsf{up}_{i-1,i+1}\left(\frac{Q_{i-1,i}(Z_{i-1}, X_{i+1}) - Q_{i-1,i}(Z_{i-1}, X_i)}{X_{i+1} - X_i}\right).$$

The quotient can be computed in $\mathcal{O}(\delta_{i-1}\,d_{i+1}^2)$. Next we apply $\mathrm{up_{i\text{-}1}}$ coefficientwise on a polynomial of degree d_{i+1} in Z_{i+1} — this is possible, since we know $Q_{i-1,i}$, so this costs $\mathcal{O}(\mathsf{M}(n)\,\mathsf{M}(\delta_i)\,d_{i+1})$. To summarize, we can compute $Q_{i,i+1}$ from $Q_{i-1,i}$ in time $\mathcal{O}(\mathsf{M}(n)\,\mathsf{M}(\delta_{i+1}))$. By the discussion on the function $S(m, n, \ell)$, with here $\ell = 0$, the *total* cost from $Q_{0,1} = Q_1$ to $Q_{i,i+1}$ is $\mathcal{O}(\mathsf{M}(n)\,\mathsf{M}(\delta_{i+1}))$. □

Corollary 7.11. *Suppose that $\mathfrak{P}_1$, ..., $\mathfrak{P}_{m-1}$ are known and compatible. If the characteristic of $\Bbbk$ is either 0 or at least δ_m, then for any $P \in \mathbb{A}_m$, we can compute $\mathcal{X}_{P,\mathbb{A}_m}$ in time $\mathcal{O}(n^{(\omega+1)/2}\,m\,\mathsf{M}(\delta_m))$.*

If $P = \Lambda$ is a primitive linear form in $\mathbb{A}_m$, compatible with the previous ones, we can compute the corresponding parametrizations in the same expected amount of time.

Proof. The first part is obvious, as the dominant term from Proposition 7.9 comes from Step 3.a.

When $P = \Lambda$ is primitive, we will write as usual Q_m instead of $\mathcal{X}_{P,\mathbb{A}_m}$. Using the discussion in Subsection 7.2.2, we can compute Q_m *and* the last parametrization $S_{m,m}$ of $\mathfrak{P}_m$ in the same cost. The other parametrizations are obtained from $\mathfrak{P}_{m-1}$by $S_{m,j} = \mathsf{up}_{m-1}(S_{m-1,j})$ for $j < m$. This is done using Proposition 7.10, since all that is required for algorithm up_{m-1} are Q_m and $S_{m,m}$. So all other parametrizations cost $\mathcal{O}(m\,\mathsf{M}(n)\,\mathsf{M}(\delta_m))$, which is not dominant. □

Proof. (of Theorem 7.1) We will give here the complexity estimate for computing $\mathfrak{P}_1, ..., \mathfrak{P}_m$ – once they are known, computing the characteristic polynomial of an arbitrary P is done using the corollary above.

We need to pick $\Lambda := 1 + \lambda_2\, X_2 + \cdots + \lambda_m\, X_m \in \mathbb{A}_m$ primitive such that its restrictions $\Lambda_i := 1 + \lambda_2\, X_2 + \cdots + \lambda_i\, X_i$ to fewer variables are still primitive. As per the assumption on the characteristic of $\Bbbk$, we pick the coefficients $\lambda_2, ..., \lambda_m$ in $\{1, ..., 2\,\delta_m^2\}$. By the remark in Subsection 7.2.2, for $2 \leqslant i \leqslant m$, Λ_i is not primitive for $\mathbb{A}_i$ with probability at most $\delta_i^2/4\,\delta_m^2$. Because of the inequality

$$\sum_{2 \leqslant i \leqslant m} \frac{\delta_i^2}{\delta_m^2} \leqslant \sum_{i \in \mathbb{N}} \frac{1}{(n!)^2} < 2.5,$$

the probability of *all* Λ_i being primitive is at least 0.375. Thus, on average, we have to pick a finite number of Λ.

Our algorithm first picks Λ as explained above. We assumed in Subsection 7.4.2 that the representation $\mathfrak{P}_1$ ought to be associated to $\Lambda_1 = X_1$, so that $\mathfrak{P}_1 = (f(Z_1), Z_1)$. Assume now that $\mathfrak{P}_1, ..., \mathfrak{P}_{i-1}$ are known. Using the first point in the previous corollary, we compute $\mathcal{X}_{\Lambda_i,\mathbb{A}_i}$ and we test whether this polynomial is squarefree. If not, we start all over from a new Λ. Otherwise, we continue with the second point in the corollary, to deduce $\mathfrak{P}_i$.

The dominant cost comes from applying the corollary. Since we expect to pick finitely many Λ, the expected cost is $\mathcal{O}(\sum_{i \leqslant m}\, n^{(\omega+1)/2}\; i\; \mathsf{M}(\delta_i))$. This is $\mathcal{O}(n^{(\omega+1)/2}\; m\; \mathsf{M}(\delta_m))$, in view of our discussion on the function $S(m,\, n,\, \ell)$, with here $\ell = 1$. This concludes the proof of our main theorem. □

Improvements given in [Leh97, RV99] to take into account predictable multiplicities in the successive resultants can be applied here as well; however, it is unclear to us how they would impact the complexity analysis.

Our last remark concerns examples from the previous section. We mentioned there some issues with the application of Proposition 7.4 (and its multiplicative version) to the polynomial $X_1\, X_2\, X_3 + X_4$, as we could not apply that proposition recursively to the polynomial $(1 + X_1\, X_2) \times X_2$. The result above shows that we can compute the characteristic polynomial of $(1 + X_1\, X_2) \times X_2$ in time $\mathcal{O}(n^{(\omega+1)/2}\,\mathsf{M}(\delta_2)) = \mathcal{O}(\mathsf{M}(\delta_4))$. As a result, we are thus able to complete the whole computation for P in quasi-linear time $\mathcal{O}(\mathsf{M}(\delta_4))$ as well. The same holds for $X_1\, X_2 + X_3\, X_4$.

7.5 Implementation and timings

Our algorithms were implemented in Magma 2.17.1; we report here on some experiments dedicated to computations in the case $m = n$, that is, in $\mathbb{A}_n$. Timings were measured on one core of a Intel Xeon at 2.27GHz with 74Gb RAM.

When $m = n$, although the complexity of CharNS is not quasi-linear (due to a 2^n overhead), it usually does better than algorithm CharResultant. A first reason is that for the former, the constant in the big-O is mild (we do only a few multiplications at each step). Besides, some other ideas are used in our code. Different recursive calls have often computations in common, so we use memoization. We also make use of symmetries: if Λ has a large stabilizer, as explained in Section 7.2, we can reduce the number of Newton sums we need to compute its characteristic polynomial. We usually attempt to pick favorable Λ: a good strategy is to take $\Lambda = \sum_{1 \leqslant i \leqslant n-1} i\, X_{n-i}$, for which the linear forms over $\mathbb{A}_{n-2}$ (which are the most expensive) have repeated coefficients.

In the following table, we take $\Bbbk = \mathbb{F}_p$, with p a 28 bit prime; we give timings to compute a univariate representation of $\mathbb{A}_n$. We are not aware of other available implementations for this problem in Magma, so we compared our algorithm with the Magma Gröbner basis functions. Our algorithm is tailored for computations in $\mathbb{A}_n$, so it is at an advantage compared to generalist functions; on the other hand, Magma's Gröbner basis functions use highly optimized C code. Despite an extra 2^n factor in the cost analysis, algorithm CharNS performs very well for this computation.

n		4	5	6	7	8
Time	Gröbner	0.001	0.03	5.8	1500	>6h
(sec)	CharNS	0.005	0.05	0.52	6.8	100

Table 7.1. Timings of computation of univariate representations

Next, we discuss the cost of basic arithmetic in $\mathbb{A}_n$, comparing in particular univariate operations to arithmetic modulo the Cauchy modules. Several Magma constructions exist for this purpose; we report on the most efficient solutions we found. As a conclusion, for an operation such as inversion, even with the overhead of lift-up and push-down, it pays off to convert to a univariate representation.

n		5	6	7	8
Time (sec)	Up	0.008	0.1	2	40
	Down	0.01	0.1	1.4	25
	Univ. $\times$	$40\mu s$	0.0005	0.006	0.06
	Univ. $\div$	0.002	0.028	0.29	4.5
	Magma $\times$	0.003	0.085	4	170
	Magma $\div$	0.1	28	>30min	>6h

Table 7.2. Timings of arithmetic in $\mathbb{A}_n$

Finally, we focus computing $\mathcal{X}_{P,\mathbb{A}_n}$, for a generic polynomial P. The best alternative we could find comes from [Sho94] and is written "Shoup" in the table. This algorithm uses univariate arithmetic; for it to be applicable, we must already know a univariate representation of $\mathbb{A}_n$, and the input must be written on the corresponding univariate basis. The complexity of "Shoup" is higher than that of CharResultant, but the algorithm is simpler and relies on fast built-in MAGMA code; as a result, it outperforms CharResultant. If the input P is a linear form in $X_1, ..., X_n$, CharNS is actually faster than both, as showed in the first table.

n		4	5	6	7	8
Time	Shoup	0.001	0.01	0.23	6.8	200
(sec)	CharResultant	0.03	0.24	2.6	45	1100

Table 7.3. Timings of computation of $\mathcal{X}_{P,\mathbb{A}_n}$ for generic polynomials P

Annexe A
Lifting of fundamental invariants

This short appendix is dedicated to prove a useful result in invariant theory, that we obtained while of writing Chapter 7: it shows that so-called fundamental invariants of finite group actions always specialize well modulo all primes, except a few exceptions known in advance. This is a rare phenomenon in computer algebra, since as a rule of thumb, for non-linear systems, the primes of "bad reduction" cannot be determined in any straightforward manner.

This result has practical implications. For example, in order to compute rational primary invariants, it is sufficient to compute primary invariants modulo p. Then the lifting of primary invariants to rational coefficients is trivial.

Private communications with H. E. A. Campbell, D. Wehlau and M. Roth revealed that this result was known to them, but as far as we know, it has not appeared in print before. We chose to include it in this thesis, since it could find practical applications — to the best of our knowledge, software such as Magma [BCP97] do not make use of this kind of result in their algorithms for computing invariant rings.

This is a joint work with É. Schost. We thank N. Thiéry for providing a simplification of our original proof.

A.1 Basic definitions

Let $n \in \mathbb{N}$ and let k be a field; we write $k[\underline{X}] := k[X_1, ..., X_n]$. We consider a finite group G and a faithful representation (that is, an injective group morphism)

$$\rho: G \to \mathrm{GL}_n(k).$$

The group G induces a right-action on $k[\underline{X}]$ given by

$$\forall g \in G, \quad \forall P \in k[\underline{X}], \quad P^g(X_1, ..., X_n) = P(g(X_1, ..., X_n)).$$

We denote by $k[\underline{X}]^G$ the k-algebra of invariant polynomials; this algebra is finitely generated and graded by total degree. The Reynolds operator

$$\mathcal{R}_G(P) = \frac{1}{|G|} \sum_{g \in G} P^g \in k[\underline{X}]^G$$

is a $k[\underline{X}]^G$-module homomorphism, and induces a projection $k[\underline{X}] \to k[\underline{X}]^G$ (see e.g. [Stu93, Proposition 2.1.2]).

Some algebraically independent homogeneous invariants $\Pi_1, ..., \Pi_n \in k[\underline{X}]^G$ are called *primary invariants* if $k[\underline{X}]^G$ is a finitely generated $k[\Pi_1, ..., \Pi_n]$-module. If the characteristic of k does not divide $|G|$, then $k[\underline{X}]^G$ is a Cohen-Macaulay ring (see e.g.[DK02, Theorem 3.4.1]); in that case, if $\Pi_1, ..., \Pi_n$ are primary invariants, then $k[\underline{X}]^G$ is actually a free $k[\Pi_1, ..., \Pi_n]$-module.

The elements S_1, ..., S_r of any $k[\Pi_1, ..., \Pi_n]$-module basis of $k[\underline{X}]^G$ are called *secondary invariants*, and the following direct sum is then called *Hironaka decomposition*:

$$k[\underline{X}]^G = \bigoplus_{j=1}^{r} k[\Pi_1, ..., \Pi_n]\, S_j.$$

We call fundamental invariants the union of primary and secondary invariants.

A.2 Main result

Let us now describe the context of our main theorem. We consider a local domain A, its fraction field $\mathbb{K} = \mathrm{Frac}\, A$ and a maximal ideal $\mathfrak{m}$ in A, with residual field $\Bbbk = A/\mathfrak{m}$. We will suppose that the characteristic of $\Bbbk$ does not divide $|G|$. We fix a faithful representation $\rho: G \hookrightarrow \mathrm{GL}_n(\mathbb{K})$.

We denote by $\bar{G} \subset \mathrm{GL}_n(\Bbbk)$ the group obtained by reducing all elements of G modulo $\mathfrak{m}$ (where we consider G as a matrix group), assuming that this reduction makes sense; the group morphism $G \to \bar{G}$ is then onto. The group morphism $G \to \overline{G}$ is surjective. Let H be its kernel, so that $\overline{G} \simeq G/H$. We fix a set of representatives $S \subset G$ of each right-class over H.

Lemma A.1. *The mapping $\varphi: A[\underline{X}]^G \to \Bbbk[\underline{X}]^{\bar{G}}$ of reduction modulo $\mathfrak{m}$ is a well-defined, surjective ring morphism.*

Proof. We first prove that the image of an invariant polynomial is invariant. For $P \in A[\underline{X}]$, we have

$$\begin{aligned}
\varphi(\mathcal{R}_G(P)) &= \overline{\mathcal{R}_G(P)} &&= \overline{\frac{1}{|G|}\sum_{g\in G} P^g} \\
&= \frac{1}{|G|}\sum_{g\in G} \left(\overline{P^g}\right) &&= \frac{1}{|G|}\sum_{s\in S}\sum_{h\in H} \bar{P}^{\bar{s}\bar{h}} \\
&= \frac{|H|}{|G|}\sum_{s\in S} \bar{P}^{\bar{s}} &&= \frac{1}{|\bar{G}|}\sum_{g'\in \bar{G}} \bar{P}^{g'} \\
&= \mathcal{R}_{\bar{G}}(\bar{P}).
\end{aligned}$$

Thus, if $P \in A[\underline{X}]^G$, we have $\varphi(P) = \varphi(\mathcal{R}_G(P)) = \mathcal{R}_{\bar{G}}(\varphi(P))$ and so $\varphi(P) \in \Bbbk[\underline{X}]^{\bar{G}}$. To prove surjectivity, let $P \in A[\underline{X}]$ be such that $\bar{P} \in \Bbbk[\underline{X}]^{\bar{G}}$. Then, we have

$$\varphi(\mathcal{R}_G(P)) = \mathcal{R}_{\bar{G}}(\bar{P}) = \bar{P}\,,$$

and surjectivity follows, since $\mathcal{R}_G(P) \in A[\underline{X}]^G$. □

We can then state our main result.

Theorem A.2. *Let $\overline{\Pi_1}, ..., \overline{\Pi_n}$ and $\bar{S}_1, ..., \bar{S_r}$ be respectively primary and secondary invariants of $\Bbbk[\underline{X}]^{\bar{G}}$. Using Lemma A.1, we can assume that $\Pi_1, ..., \Pi_n, S_1, ..., S_r \in A[\underline{X}]^G$. Then, these invariants are respectively primary and secondary invariants for $\mathbb{K}[\underline{X}]^G$.*

Proof. Let $d \in \mathbb{N}$. We denote by $(\mathbb{K}[\underline{X}]^G)_d$ the homogeneous component of degree d of $\mathbb{K}[\underline{X}]^G$. Since $\overline{\Pi_1}, ..., \overline{\Pi_n}$ and $\bar{S}_1, ..., \bar{S_r}$ are fundamental invariants of $\Bbbk[\underline{X}]^{\bar{G}}$, we know from the Hironaka decomposition that the family of polynomials

$$\mathcal{F}_{\Bbbk} := \left\{ \overline{\Pi_1}^{i_1} \cdots \overline{\Pi_n}^{i_n} \bar{S_j} \middle| \deg(\bar{S_j}) + \sum_{l=1}^{n} i_l \deg(\bar{\Pi_l}) = d \right\}$$

form a $\Bbbk$-vector basis of $\left(\Bbbk[\underline{X}]^{\bar{G}}\right)_d$. We aim at proving it for $\Pi_1, ..., \Pi_n, S_1, ..., S_r$. That is we prove that the family

$$\mathcal{F}_{\mathbb{K}} := \left\{ \Pi_1^{i_1} \cdots \Pi_n^{i_n} S_j \middle| \deg(S_j) + \sum_{l=1}^{n} i_l \deg(\Pi_l) = d \right\}$$

is a $\mathbb{K}$-vector basis of the homogeneous component $(\mathbb{K}[\underline{X}]^G)_d$. Then the Hironaka decomposition follows straightforwardly and so does the fact that $\Pi_1, ..., \Pi_n, S_1, ..., S_r$ are fundamental invariants.

We start by proving that we can take a common vector basis of $\Bbbk[\underline{X}]^{\bar{G}}_d$ and $\mathbb{K}[\underline{X}]^G_d$. By Lemma A.1, the projection $\varphi\colon A[\underline{X}]^G \to \Bbbk[\underline{X}]^{\bar{G}}$ is surjective. So let $\mathcal{B} = \{b_1, ..., b_r\}$ be such that $\varphi(\mathcal{B}) := (\varphi(b_1), ..., \varphi(b_r))$ is a $\Bbbk$-vector space basis of $\Bbbk[\underline{X}]^{\bar{G}}_d$. By Nakayama's Lemma (see e.g. [AM69, Proposition 2.8]), we know that $\mathcal{B}$ generates $A[\underline{X}]^G_d$ as an A-module. For any $P \in \mathbb{K}[\underline{X}]^G$, there exists $a \in A$ such that $a\,P \in A[\underline{X}]^G$. So we can deduce that $\mathcal{B}$ generates the $\mathbb{K}$-vector space $\mathbb{K}[\underline{X}]^G_d$.

We will prove in Lemma A.3 below that $(\Bbbk[\underline{X}]^{\bar{G}})_d$ and $\mathbb{K}[\underline{X}]^G_d$ have the same dimension, so $\mathcal{B}$ is a $\mathbb{K}$-vector basis of $\mathbb{K}[\underline{X}]^G_d$.

We denote by M the matrix at coefficient in A whose columns are the coordinates of the family $\mathcal{F}_{\mathbb{K}}$ in the basis $\mathcal{B}$. Since $\mathcal{F}_{\Bbbk}$ is a basis of $(\Bbbk[\underline{X}]^{\bar{G}})_d$, we know that the reduction of M modulo $\mathfrak{m}$ is invertible. So the determinant of M is non-zero modulo $\mathfrak{m}$, and consequently it is non-zero in $\mathbb{K}$. Since M is invertible, $\mathcal{F}_{\mathbb{K}}$ is a basis of $\mathbb{K}[\underline{X}]^G_d$, which concludes the proof. □

Recall that the Hilbert series $\mathcal{H}(A, t)$ of a graded k-algebra $A = \bigoplus_{n\in\mathbb{N}} A_n$ is given by

$$\mathcal{H}(A, t) = \sum_{n\in\mathbb{N}} \dim_k(A_n)\, t^n.$$

The last missing ingredient is the following lemma about the Hilbert series associated to $k[\underline{X}]^G$ and $k[\underline{X}]^{\bar{G}}$.

Lemma A.3. *The following equality holds:*

$$\mathcal{H}(\mathbb{K}[\underline{X}]^G, t) = \mathcal{H}(\Bbbk[\underline{X}]^{\bar{G}}, t).$$

Proof. Note first that the eigenvalues of the elements of G are all $|G|$th-roots of unity. Let us then fix a group isomorphism between the $|G|$th-roots of unity is a suitable extension of $\Bbbk$ and those in $\mathbb{C}$. For $g \in \bar{G}$, we write $\det^0(1 - t\,g) = (1 - t\,\lambda_1) \cdots (1 - t\,\lambda_n) \in \mathbb{C}[t]$, where $\lambda_1, ..., \lambda_n$ are the images of its eigenvalues in $\mathbb{C}$ (this is independent of the isomorphism we chose).

Then, *Molien's formula* (see e.g. [DK02, Theorem 3.2.2]) allows us to compute $\mathcal{H}(k[\underline{X}]^G, t)$ and $\mathcal{H}(k[\underline{X}]^{\bar{G}}, t)$ as follows:

$$\mathcal{H}(k[\underline{X}]^G, t) = \frac{1}{|G|} \sum_{g \in G} \frac{1}{\det(1 - t\,g)}$$

and

$$\mathcal{H}(k[\underline{X}]^{\bar{G}}, t) = \frac{1}{|\bar{G}|} \sum_{g \in \bar{G}} \frac{1}{\det^0(1 - t\,g)}.$$

Now, by definition of $\det^0$, for all $g \in G$, we have

$$\det^0(1 - t\,g) = \det(1 - t\,g) = \det^0(1 - t\,\bar{g}).$$

Then, we get

$$\begin{aligned} \mathcal{H}(\mathbb{K}[\underline{X}]^G, t) &= \frac{1}{|G|} \sum_{g \in G} \frac{1}{\det^0(1 - t\,g)} = \frac{1}{|G|} \sum_{g \in G} \frac{1}{\det^0(1 - t\,\bar{g})} \\ &= \frac{|H|}{|G|} \sum_{s \in S} \frac{1}{\det^0(1 - t\,\bar{s})} = \frac{1}{|\bar{G}|} \sum_{g' \in \bar{G}} \frac{1}{\det^0(1 - t\,g')} \\ &= \mathcal{H}(\Bbbk[\underline{X}]^{\bar{G}}, t), \end{aligned}$$

so the lemma is proved. □

For applications, the simplest case is when the matrices of G have integer, or possibly rational entries. Let d be the lcm of all denominators arising in the entries of the elements in G and let p be a prime that does not divide $d|G|$. We let A be the local ring $\mathbb{Z}_{(p)}$, $\mathbb{K} = \mathbb{Q}$ and $\mathfrak{m} = (p)$. Then, to compute primary and secondary invariants for $\mathbb{Q}[\underline{X}]^G$, it is enough to compute such invariants modulo p, for $\mathbb{F}_p[\underline{X}]^{\bar{G}}$.

More generally, one can consider the case of number fields. Consider a faithful representation $\rho: G \hookrightarrow \mathrm{GL}_n(\mathbb{Q}(\theta))$, where θ is algebraic over $\mathbb{Q}$.

Let $\mathcal{O}$ be the ring of integers of $\mathbb{Q}(\theta)$; then, each $c \in \mathbb{Q}(\theta)$ can be written as $c = p/q$, with $p \in \mathcal{O}$ and $q \in \mathbb{N}$. We let d be the lcm of all denominators arising in this fashion for the elements of G, and we take p a prime that does not divide d, nor $|G|$. The ideal $p\mathcal{O} \subset \mathcal{O}$ may not be prime, so we choose a prime (hence maximal) ideal $\mathcal{P}$ that contains it; then, all coefficients of the elements of G are in the local ring $\mathcal{O}_{\mathcal{P}}$.

We can then apply Theorem A.2 to the localization $A = \mathcal{O}_{\mathcal{P}}, \mathbb{K} = \mathbb{Q}(\theta), \Bbbk = \mathbb{F}_{p^m} = \mathcal{O}_{\mathcal{P}}/\mathcal{P}$. To obtain primary and secondary invariants for $\mathbb{Q}(\theta)[\underline{X}]^G$, it is then enough to compute them modulo p, for $\mathbb{F}_{p^m}[\underline{X}]^{\bar{G}}$.

Using slightly more advanced results from (see e.g. [Ser78, Chapter 2, Theorem 6]), one can prove that the number field case includes most other cases: if G is a finite group and k is a field of characteristic zero, then any representation of G on k is isomorphic (over k) to a representation on $\bar{\mathbb{Q}} \cap k$.

Annexe B
Introduction
(translated into English)

Nowadays, the increasing performance of computers is used, among other things, to refine the accuracy of known solutions to problems or to tackle challenges with larger inputs. However, it is common in Computer Science that the running time of an algorithm increases many times faster than the desired precision for the solutions. An efficient strategy to remedy this fact is to cut one large problem into many small parts, solve them and reconstruct the solution of the original problem.

In the context of Computer Algebra, the Chinese remainder theorem gives such a tool. Split a problem into many identical problems modulo coprime integers $n_1, \ldots, n_r$. Then the Chinese remainder theorem allows the reconstruction of the solution of the main problem from the solutions of the split problems. This scheme of computation, when applicable, allows one to compute in time proportional to the precision, plus a multi-modular reconstruction.

Multi-modular techniques can be used in association to p-adic lifting. Take one of the coprime elements, e.g. n_1, of the form $n_1 = p^\ell$. Then reduce the size of the problem by solving it only modulo p. The reconstruction of the solution modulo p^ℓ from the one modulo p is called *p-adic lifting*. Hence, to solve a problem modulo a power p^ℓ, we have to solve *one* small problem and then apply a p-adic lifting. As the lifting may be easier to compute than to solve the whole problem, we may spare a lot of time by using the p-adic lifting technique.

This thesis is mainly dedicated to the fast computation of p-adic lifting by a novel type of algorithms, the relaxed algorithms.

* * *

My work is in the line of a series of papers dealing with relaxed algorithms initiated by van der Hoeven [Hoe97, Hoe02, Hoe03, Hoe07, Hoe09] for powers series and adapted by Berthomieu, Lecerf and van der Hoeven [BHL11] to the case of general p-adic rings. Roughly speaking, a p-adic a is an infinite sequence of coefficients $(a_i)_{i\in\mathbb{N}}$ that we write $a := \sum_{i\in\mathbb{N}} a_i p^i$.

Relaxed algorithms are a special case of on-line algorithms. On-line algorithms were introduced by Hennie [Hen66] in the Turing model. An *on-line algorithm* with p-adic inputs is an algorithm that reads the p-adic coefficients of the input one by one, and outputs the nth coefficient of the output before reading the $(n+1)$th coefficient of the input.

For instance, the multiplication of two p-adics by an on-line algorithm takes, at first glance, quadratic time in the precision. But a quasi-optimal on-line multiplication algorithm exists and can be found in [FS74, Sch97, Hoe97].

The major advantage of on-line algorithms is that they enable the lifting of *recursive* p-adics. A recursive p-adic y is a p-adic that satisfies $y = \Phi(y)$ for an operator Φ such that the nth coefficient of the p-adic $\Phi(y)$ does not depend on the coefficients of order greater or equal to n of y. As a consequence, y can be computed recursively from its first coefficient y_0 and Φ.

One can find in [Wat89, Hoe02] an algorithm that computes y from its fixed point equation $y = \Phi(y)$. A key aspect of this algorithm is that *its cost is the one of evaluating* Φ by an on-line algorithm. Using the fast on-line multiplication algorithm of [Hoe97], it leads to an efficient framework for computing with recursive p-adics. As van der Hoeven was apparently not aware of previous work on on-line algorithms, he called his algorithm "relaxed multiplication algorithm" and the subsequent algorithms for recursive p-adics were called relaxed algorithms. Therefore we can retrospectively define a *relaxed algorithm* to be a fast on-line algorithm. Relaxed algorithms are also related to lazy algorithms, which can be seen as on-line algorithms that try to minimize the cost at each step. To sum up, relaxed and lazy algorithms are two instances of on-line algorithms, with the first minimizing the cost globally and the second locally. These definitions, and their adaptation to general p-adic rings [BHL11], will be used for now on.

A main contribution of this thesis is to use relaxed algorithms in the context of resolution of equations by p-adic lifting. This context is applicable to standard and important systems of equations: linear, algebraic and differential.

The on-line framework for recursive p-adic lifting has to be compared with the other general framework for resolution of equations by p-adic lifting. Whenever a p-adic y is given by an implicit equation $f(y) = 0$ whose derivative in y is invertible, the Hensel-Newton operator can be applied to compute y at any precision given its first coefficient y_0. This operator was introduced by Newton in [New36] and adapted to the case of p-adic integers by Hensel in [Hen18]. There exist situations where the Newton operator do not apply in a straightforward manner. However, the underlying principle can often be adapted (see Chapters 4, 6).

As we will see on many occasions throughout this thesis, on-line lifting algorithms perform, asymptotically in the precision, less on-line multiplications than off-line algorithms do off-line multiplications. For example, on-line algorithms reduce the cost due to inverting the Jacobian matrix in Newton iteration. In return, on-line multiplication costs more than off-line multiplication. So we will implement most of our algorithms and compare them in practice.

* * *

This thesis explores the relaxed p-adic lifting of the solutions of many different types of systems. Part I introduces relaxed algorithms and their application to recursive p-adics. In Chapter 1, after giving the definition of on-line algorithms, we recall several fast on-line, or relaxed, multiplication algorithms for p-adics and introduce a new one. Then in Chapter 2, we introduce a framework in which recursive p-adics can be computed by on-line algorithms. We will use this framework in Parts II and III of this thesis to give new on-line p-adic lifting algorithms for solutions of different types of systems.

Part II is focused on linear systems. We start in Chapter 3 by linear algebra over p-adics and solve linear systems taking into consideration the representation of the matrices. Next, we provide algorithms computing power series solutions of a large class of differential or (q)-differential systems in Chapter 4.

In increasing order of generality, Part III is devoted to the p-adic lifting of solutions of algebraic systems. Chapter 5 gives an on-line algorithm for the p-adic lifting of regular roots of any algebraic system. Chapter 6 deals with the lifting of univariate representations and triangular sets by on-line algorithms.

Part IV is dedicated to the special case of the ideal of symmetric relations. In this setting, before the lifting step, the computation of a univariate representation at precision 0 was already an issue. We give in Chapter 7 a quasi-optimal algorithm to compute this univariate representation, and use it to obtain efficient algorithms to compute in the corresponding quotient algebra.

In Appendix A, we prove that the lifting of any so-called fundamental invariants modulo p to rational coefficients is trivial.

Contributions To summarize, the contributions of this thesis are:

1. A new relaxed multiplication algorithm of p-adics;
2. A thorough complexity analysis of several relaxed multiplication algorithms;
3. An accurate framework to compute recursive p-adics by on-line algorithms;
4. A relaxed linear system solver over p-adics;
5. Two new lifting algorithms of power series solutions of singular (q)-differential equations: one relaxed algorithm, and one off-line which adapts the Newton iteration;
6. A new relaxed lifting algorithm for p-adic regular root of algebraic systems;
7. More generally, a new relaxed lifting algorithm for triangular sets and univariate representations;
8. Next, we give the first quasi-linear algorithms to compute a univariate representation of the universal decomposition algebra on finite fields and to compute characteristic polynomials of its elements;
9. Finally, we show that in order to compute rational fundamental invariants, it is sufficient to compute fundamental invariants modulo p.

Publications The contributions 3, 4 and 6 were published in the proceedings of *ISSAC'12* with J. Berthomieu [BL12]. Their presentation in this thesis contains additional details, proofs and examples. Moreover, the application of the relaxed linear algebra solver of item 4 over p-adics for structured matrices is a joint work in progress with É. Schost. The contributions of item 5 is a joint work with A. Bostan, M. Chowdhurry, B. Salvy and É. Schost that was published in the proceedings of *ISSAC'12* [BCL+12]. Finally, the contributions of item 8 is a joint work with É. Schost, published in the proceedings of *ISSAC'12* [LS12].

Awards I received the "Best Student Paper Award" for the paper [LS12] at the conference *ISSAC'12*. I also received the best poster award from the "Fachgruppe Computer Algebra" for the poster [LMS12] in collaboration with E. Mehrabi and É. Schost.

B.1 Relaxed algorithms for multiplication

The section introduces the notion of on-line and relaxed algorithms on general p-adic rings. Let R be a commutative ring with unit. Given a proper principal ideal (p) with $p \in R$, we write R_p for the completion of the ring R for the p-adic valuation. An element $a \in R_p$ is called a p-adic. To get a unique writing of elements in R_p, let us fix a subset M of R such that the projection $\pi: M \to R/(p)$ is a bijection. Then, any element $a \in R_p$ can be uniquely written $a = \sum_{i \in \mathbb{N}} a_i p^i$ with coefficients $a_i \in M$.

Two classical examples are the formal power series ring $\Bbbk[[X]]$, which is the completion of the ring of polynomials $\Bbbk[X]$ for the ideal (X) and the ring of p-adic integers $\mathbb{Z}_p$, which is the completion of the ring of integers $\mathbb{Z}$ for the ideal (p), with p a prime number. For $R = \mathbb{Z}$, we take $M = \{-(p-1)/2, \ldots, (p-1)/2\}$ if $p \neq 2$ and $M = \{0, 1\}$ for $p = 2$. For $R = \Bbbk[X]$, we take $M = \Bbbk$.

We can now give the definition of on-line algorithm on p-adics introduced by [Hen66].

Definition B.1. ([Hen66, FS74]) *Let us consider a Turing machine which computes a function f on sequences, where $f\colon \Sigma^* \times \Sigma^* \to \Delta^*$, Σ and Δ are sets. The machine is said to compute f* on-line *if for all input sequences $a = a_0 a_1 \ldots a_n$, $b = b_0 b_1 \ldots b_n$ and corresponding outputs $f(a, b) = c_0 c_1 \ldots c_n$, with $a_i, b_j \in \Sigma$, $c_k \in \Delta$, it produces c_k before reading either a_j or b_j for $0 \leqslant k < j \leqslant n$.*

The machine computes f half-line *(with respect to the first argument) if it produces c_k before reading a_j for $0 \leqslant k < j \leqslant n$. The input a will be referred to as the* on-line argument *and b as the* off-line argument*.*

The remaining part of the chapter is devoted to the presentation of fast on-line algorithms for the multiplication of p-adics. These algorithms are made of suitable calls to off-line multiplication algorithms on finite precision p-adics. We start by recalling the state-of-the-art of multiplication algorithms on polynomials and integers, which correspond respectively to finite precision power series and p-adic integers. We also give a special focus on the existing algorithms for middle product and short product.

With these notions at hand, we recall the following existing relaxed multiplication algorithms for p-adics. The first quasi-optimal on-line algorithm for the multiplication was presented in [FS74] for integers, then in [Sch97] for real numbers and finally in [Hoe97] for power series. This latter algorithm was adapted to a semi-relaxed (or half-line) multiplication algorithm in [FS74, Hoe03]. A first improvement of the semi-relaxed product by a constant factor is presented in [Hoe03].

Our contribution is to present a new relaxed algorithm for the multiplication using *middle* and *short* product (when R_p supports such operations), that improves by a constant factor the previous ones. Moreover, we give for the first time a thorough analysis of the arithmetic complexity of all these multiplication algorithms. Finally, we show some timings to confirm the good behavior of relaxed algorithms using middle product.

From now on, we use the following notations. For any p-adic $a = \sum_{i \in \mathbb{N}} a_i p^i$, the length $\lambda(a)$ of a is defined by $\lambda(a) := 1 + \sup(i \in \mathbb{N} \mid a_i \neq 0)$ if $a \neq 0$ and $\lambda(0) = 0$. The cost of multiplying two p-adics of length N by an off-line (resp. an on-line algorithm) is denoted by $\mathsf{I}(N)$ (resp. $\mathsf{R}(N)$) in our complexity model specified in Section 1.1.2. Next, we denote by $\mathsf{M}(N)$ the arithmetic complexity of the multiplication of two polynomials of length N.

B.2 Recursive p-adics

The study of on-line algorithms is motivated by their efficient implementation of recursive p-adics. It was first spotted in [Wat89] that the lifting of recursive power series is well-suited to lazy algorithms. It gave at the time a very convenient framework to compute recursive power series but not yet efficient.

This fact was rediscovered in [Hoe02] for general on-line algorithms. Together with the fast on-line multiplication algorithm of [Hoe97], it led to an efficient framework for computing with recursive power series.

A recursive p-adic y is a p-adic that satisfies $y = \Phi(y)$ for an operator Φ such that we have the equality between the nth coefficients $\Phi(y)_n = \Phi(y + p^n a)_n$ for any $a \in R_p$. As a consequence, y is uniquely determined by its first coefficient y_0 and Φ.

In this chapter, we recall the method from [Hoe02] which, from an on-line algorithm that evaluates the function Φ, outputs the coefficients of the recursive p-adic y one by one. However this method does not always work as is.

Our contribution is to raise and solve the problem to find which functions Φ make the method work. This issue was never mentioned before in the literature.

In order to fix this problem, we introduce the new notion of *shifted algorithms*. An integer called the shift is associated to any p-adic input of an algorithm and measure which coefficients of this input are read when the algorithm produces the nth coefficient of the output. For example, an algorithm is on-line if and only if its corresponding shift is non-negative.

Then we define a *shifted algorithm* to be an algorithm that has a positive shift. Now we have the tools to state the next fundamental proposition.

Proposition. *Let y be a recursive p-adic and Ψ be a shifted algorithm such that $y = \Psi(y)$. Then, the p-adic y can be computed at precision N in the time necessary to evaluate Ψ at y at precision N.*

Hence the cost to compute a recursive p-adic is the same that the cost of verifying it. This latter proposition is the cornerstone of complexity estimates regarding recursive p-adics and will be used in Chapters 3, 4, 5 and 6.

B.3 Linear algebra over p-adics

We introduce an algorithm based on the p-recursive framework of Chapter 2, which can in principle be applied to all representation of matrices (dense, sparse, structured, ...). We focus on two important cases, *dense* and *structured* matrices, and show how our algorithm can improve on existing techniques in these cases.

We consider a linear system of the form $A = B \cdot C$, where A and B are known, and C is the unknown. The matrix A belongs to $\mathcal{M}_{r\times s}(R_p)$ and $B \in \mathcal{M}_{r\times r}(R_p)$ is invertible; we solve the linear system $A = B \cdot C$ for $C \in \mathcal{M}_{r\times s}(R_p)$. The most interesting cases are $s=1$ (which amounts to linear system solving) and $s=r$, which contains in particular the problem of inverting B. A major application of p-adic linear system solving is actually to solve systems over R (that is over e.g. the integers or the polynomials), by means of lifting techniques.

We denote by $d := \max(\lambda(A), \lambda(B))$ the length of the entries of A and B. Let N be the precision to which we require C. Thus, we will always be able to suppose that $d \leqslant N$. The case $N = d$ corresponds to the resolution of p-adic linear systems proper, whereas solving systems over R often requires to take a precision $N \gg d$. Indeed, in that case and for e.g. power series, we deduce from Cramer's formulas that the numerators and denominators of C have length $\mathcal{O}(r\,d)$, so that we take N of order $\mathcal{O}(r\,d)$ in order to make rational reconstruction possible.

Among the existing algorithms, a first algorithm is due to Dixon [Dix82]; it finds one p-adic coefficient of the solution C at a time and then updates the matrix A. On the other side of the spectrum, one finds Newton's iteration, which doubles the precision of the solution at each step. Moenck-Carter's algorithm [MC79] is a variant of Dixon's algorithm that works with p^d-adics instead of p-adics. Finally, Storjohann's high-order lifting algorithm [Sto03] can be seen as a fast version of Moenck-Carter's algorithm, well-suited to cases where $d \ll N$.

Our contribution is an algorithm to solve linear systems by means of relaxed techniques; it is obtained by proving that the entries of the solution $C = B^{-1} \cdot A$ are recursive p-adics. In other words, we show that C is a fixed point for a suitable shifted operator.

Taking for instance $s=1$, to compute C at precision N, the cost of the resulting algorithm will (roughly speaking) involve the following:

1. the inversion $\Gamma := B^{-1}$ modulo (p),
2. $\mathcal{O}(N)$ matrix-vector products using the inverse Γ, where all entries of the right-hand side vector have also length 1,
3. $\mathcal{O}(1)$ matrix-vector product using B, with a right-hand side vector whose entries are relaxed p-adics.

We will see that our algorithm is a middle point between Dixon's and Moenck-Carter's algorithms since we do the same cheap initialization modulo (p) as Dixon (see item 1) and still have the same good asymptotic behavior as Moenck-Carter (due to item 3).

The next table gives the resulting running time for the case of dense matrices, with $R_p = \mathbb{k}[[X]]$ and $s=1$ and two practically meaningful values for N, respectively $N=d$ and $N=r\,d$. It appears that for solving up to precision $N=d$, our algorithm is the fastest among the ones we compare; for $N=r\,d$, Storjohann's high-order lifting does best (as it is specially designed for such large precisions). We let ω be such that we can multiply and invert $r \times r$ matrices within $\mathcal{O}(r^\omega)$ arithmetic operations over any field.

Algorithm	$N = d$	$N = r\,d$
Dixon	$\tilde{\mathcal{O}}(r^\omega + r^2\,d^2)$	$\tilde{\mathcal{O}}(r^3\,d^2)$
Moenck-Carter	$\tilde{\mathcal{O}}(r^\omega\,d)$	$\tilde{\mathcal{O}}(r^3\,d)$
Newton iteration	$\tilde{\mathcal{O}}(r^\omega\,d)$	$\tilde{\mathcal{O}}(r^{\omega+1}\,d)$
High-order lifting	$\tilde{\mathcal{O}}(r^\omega\,d)$	$\tilde{\mathcal{O}}(r^\omega\,d)$
Our algorithm	$\tilde{\mathcal{O}}(r^\omega + r^2\,d)$	$\tilde{\mathcal{O}}(r^3\,d)$

Table. Simplified cost of solving $A = B \cdot C$ for dense matrices over $R_p = \Bbbk[[X]]$, with $s = 1$

Next, we discuss the situation for structured matrices. In a few words, in a structured matrix representation, an integer α, called the rank, is associated to any matrix $A \in \mathcal{M}_{r\times r}(R_p)$. The two main characteristics of the structured matrix A are that A can be stored by a compact data structure in space $\mathcal{O}(\alpha\,r)$ and that the matrix-vector multiplication $A \cdot V$ costs $\mathcal{O}(\alpha\,\mathsf{M}(r))$ for any vector $V \in \mathcal{M}_{r\times 1}(R_p)$. We refer to [Pan01] for a thorough presentation.

The next table recalls previously known results about solving structured linear systems and shows the running time of our algorithm. Here, d' denotes the length of the entries of the compact data structure that stores A and N is still the target precision. As before, we give simplified results for power series, $s = 1$ and $N = d'$ or $N = r\,d'$.

Algorithm	$N = d'$	$N = r\,d'$
Dixon	$\tilde{\mathcal{O}}\big(\alpha^2\,r + \alpha\,r\,d'^2\big)$	$\tilde{\mathcal{O}}\big(\alpha\,r^2\,d'^2\big)$
Moenck-Carter	$\tilde{\mathcal{O}}(\alpha^2\,r\,d')$	$\tilde{\mathcal{O}}(\alpha\,r^2\,d')$
Newton iteration	$\tilde{\mathcal{O}}(\alpha^2\,r\,d')$	$\tilde{\mathcal{O}}(\alpha^2\,r^2\,d')$
Our algorithm	$\tilde{\mathcal{O}}(\alpha^2\,r + \alpha\,r\,d')$	$\tilde{\mathcal{O}}(\alpha\,r^2\,d')$

Table. Simplified cost of solving $A = B \cdot C$ for structured matrices over $\Bbbk[[X]]$, with $s = 1$

Our algorithm is the fastest for structured matrices for both cases $N = d$ and $N = r\,d$. Note that Moenck-Carter's algorithm is equally fast, modulo a constant factor, in the second case.

Finally, we implement these algorithms and compare timings for dense linear algebra. Our implementation is available in the C++ library ALGEBRAMIX of MATHEMAGIX [HLM+02]. As an application, we solve linear systems over the integers and compare to LINBOX and IML. We show that we improve the timings for small matrices and large integers.

B.4 Power series solutions of (q)-differential equations

The aim of this chapter is to provide algorithms computing power series solutions of a large class of differential or q-differential equations or systems. Their number of arithmetic operations grows linearly with the precision, up to logarithmic terms.

We focus on the case when this equation is linear, since in many cases linearization is possible [BCO+07]. When the order n of the equation is larger than 1, we use the classical technique of converting it into a first-order equation over vectors, so we consider equations of the form

$$x^k \delta(F) = A F + C, \tag{B.1}$$

where A is an $n \times n$ matrix over the power series ring $\Bbbk[[x]]$ ($\Bbbk$ being the field of coefficients), C and the unknown F are size n vectors over $\Bbbk[[x]]$ and for the moment δ denotes the differential operator $d/d\,x$. The exponent k in (B.1) is a non-negative integer that plays a key role for this equation.

By *solving* such equations, we mean computing a vector F of power series such that (B.1) holds modulo x^N. For this, we need only compute F polynomial of degree less than $N+1$ (when $k=0$) or N (otherwise). Conversely, when (B.1) has a power series solution, its first N coefficients can be computed by solving (B.1) modulo x^N (when $k \neq 0$) or x^{N-1} (otherwise).

If $k=0$ and the field $\Bbbk$ has characteristic 0, then a formal Cauchy theorem holds and (B.1) has a unique vector of power series solution for any given initial condition. In this situation, algorithms are known that compute the first N coefficients of the solution in quasi-linear complexity: [BK78] for scalar equations of order 1 and 2, adapted in [BCO+07] for systems of equations. Also the relaxed algorithm of [Hoe02] applies to this case.

In this chapter, we extend the previous algorithms in three directions.

Singularities We deal with the case when k is positive. Cauchy's theorem and the techniques of [BCO+07] do not apply. We show in this chapter how to overcome this singular behavior and obtain a quasi-linear complexity.

Positive characteristic Even when $k = 0$, Cauchy's theorem does not hold in positive characteristic and Equation (B.1) may fail to have a power series solution (a simple example is $F' = F$). However, such an equation may have a solution modulo x^N. Our objectives in this respect are to overcome the lack of a Cauchy theorem, or of a formal theory of singular equations, by giving conditions that ensure the existence of solutions at the required precisions.

Functional equations Linear differential equations and linear difference equations can be solved by similar algorithms. For this matter, introduce $\sigma: \Bbbk[[x]] \to \Bbbk[[x]]$ a unitary ring morphism and let $\delta: \Bbbk[[x]] \to \Bbbk[[x]]$ denote a σ-derivation, in the sense that δ is $\Bbbk$-linear and that for all f, g in $\Bbbk[[x]]$, we have

$$\delta(f g) = f\,\delta(g) + \delta(f)\,\sigma(g).$$

These definitions, and the above equality, extend to matrices over $\Bbbk[[x]]$. Thus, our goal is to solve the following generalization of (B.1):

$$x^k \delta(F) = A\,\sigma(F) + C. \tag{B.2}$$

As above, we are interested in computing a vector F of power series such that (B.2) holds modulo x^N.

Concerning on-line algorithms, the techniques of [Hoe02] already apply to the positive characteristic case. At the beginning of my thesis, the tools to adapt relaxed algorithms for singular equations did not exist. Our method to deal with singular equations was discovered independently at the same time by [Hoe11]. This paper deals with more general recursive power series defined by algebraic, differential equations or a combination thereof. However, this paper does not consider the case of (q)-differential equations and works under more restrictive hypotheses.

We restrict ourselves to the following setting:

$$\delta(x) = 1, \qquad \sigma\colon x \mapsto q\,x,$$

for some $q \in \Bbbk \setminus \{0\}$. Then, there are only two possibilities:

- $q = 1$ and $\delta\colon f \mapsto f'$ (*differential* case);
- $q \neq 1$ and $\delta\colon f \mapsto \frac{f(q\,x) - f(x)}{x\,(q-1)}$ (q-*differential* case).

Seeing Eq. (B.2) as a linear system, one can obtain such an output using linear algebra in dimension $n\,N$. While this solution always works, we give algorithms of much better complexity, under some *good spectrum* assumptions related to the spectrum $\operatorname{Spec} A_0$ of the constant coefficient A_0 of A.

As in the paper [BCO+07] for the non-singular case, we develop two approaches. The first one is a divide-and-conquer method. The problem is first solved at precision $N/2$ and then the computation at precision N is completed by solving another problem of the same type at precision $N/2$. This leads us to the following result.

Theorem. *One can compute generators of the solution space of Eq. (B.2) at precision N by a divide-and-conquer approach. Assuming A_0 has good spectrum at precision N, it can be done in time $\mathcal{O}(n^\omega\,\mathsf{M}(N)\log(N))$. When either $k > 1$ or $k = 1$ and $q^i\,A_0 - \gamma_i\,\mathrm{Id}$ is invertible for $0 \leq i < N$, this drops to $\mathcal{O}(n^2\,\mathsf{M}(N)\log(N) + n^\omega\,N)$.*

This divide-and-conquer approach coincides with an on-line recursive p-adic computation under the second set of hypothesis, that is either $k > 1$ or $k = 1$ and $q^i\,A_0 - \gamma_i\,\mathrm{Id}$ is invertible for $0 \leq i < N$. We choose to present it by a divide-and-conquer approach because it will make it convenient to study the problem under more general hypotheses.

Our second algorithm behaves better with respect to N, with cost in $\mathcal{O}(\mathsf{M}(N))$ only, but it always involves polynomial matrix multiplications. Since in many cases the divide-and-conquer approach avoids these multiplications, the second algorithm becomes preferable for rather large precisions.

In the differential case, when $k = 0$ and the characteristic is 0, the algorithms in [BCO+07, BK78] compute an invertible matrix of power series solution of the homogeneous equation by a Newton iteration and then recover the solution using variation of the constant. In the more general context we are considering here, such a matrix does not exist. However, it turns out that an associated equation that can be derived from (B.2) admits such a solution. We describe a variant of Newton's iteration to solve it and obtain the following.

Theorem. *Assuming A_0 has good spectrum at precision N, one can compute generators of the solution space of Eq. (B.2) at precision N by a Newton-like iteration in time $\mathcal{O}(n^\omega \mathsf{M}(N) + n^\omega \log(n) N)$.*

To the best of our knowledge, this is the first time such a low complexity is reached for this problem. Without the good spectrum assumption, however, we cannot guarantee that this algorithm succeeds, let alone control its complexity.

B.5 Relaxed p-adic Hensel lifting for algebraic systems

This chapter can be seen as a special case of lifting of triangular set done in Chapter 6.

We are given as input a polynomial system $\boldsymbol{P} = (P_1, ..., P_r)$ in $R[Y_1, ..., Y_r]$ and a modular root $\boldsymbol{y}_0 \in (R/(p))^r$ such that $\boldsymbol{P}(\boldsymbol{y}_0) = 0$ in $(R/(p))^r$. We work under the hypothesis of Hensel's lemma, which requires that the Jacobian matrix $\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)$ is invertible. As a consequence, there exists a unique root $\boldsymbol{y} \in R^r$ of $\boldsymbol{P}$ that reduces to $\boldsymbol{y}_0$ modulo p. We say that $\boldsymbol{y}$ is the lifted root of $\boldsymbol{P}$ from $\boldsymbol{y}_0$.

Our work consists in transforming these implicit polynomial equations into recursive equations. Therefore we can use the relaxed framework for p-adic numbers to lift a modular root $\boldsymbol{y}_0$ over $R/(p)$ to a p-adic root $\boldsymbol{y}$ over R_p. Our results on the transformation of implicit equations to recursive equations were discovered independently at the same time by [Hoe11].

For the sake of simplicity, we present here only the asymptotic complexities when the p-adic precision N tends to infinity, that is $f(n, L, d, N) = \mathcal{O}_{N\to\infty}(g(n, L, d, N))$ if there exists $K_{n,L,d} \in \mathbb{R}_{\geqslant 0}$ such that for all $N \in \mathbb{N}$, $f(n, L, d, N) \leqslant K_{n,L,d}\, g(n, L, d, N)$.

Let us start by enunciating the result for dense univariate polynomials.

Proposition. *Given a polynomial P of degree d in dense representation and a modular simple root y_0, we can lift y_0 at precision N in time $(d-1)\,\mathsf{R}(N) + \mathcal{O}_{N\to\infty}(N)$.*

In comparison, Newton iteration lifts y at precision N in time $(3\,d+4)\,\mathsf{I}(N) + \mathcal{O}_{N\to\infty}(N)$ (see [GG03, Theorem 9.25]). So the first advantage of our on-line algorithm is that it does asymptotically less on-line multiplications than Newton iteration does off-line multiplications. Also, we can expect better timings from the on-line method for the Hensel lifting of y when the precision N satisfies $\mathsf{R}(N) \leqslant 3\,\mathsf{I}(N)$.

Let us now deal with multivariate polynomial systems.

Theorem. *Let $\boldsymbol{P} = (P_1, ..., P_r)$ be a polynomial system in $R[Y_1, ..., Y_r]^r$ in dense representation, satisfying $d \geqslant 2$ where $d := \max_{1\leqslant i,j\leqslant r}(\deg_{X_j}(P_i)) + 1$, and let $\boldsymbol{y}_0$ be an approximate zero of $\boldsymbol{P}$.*

Then we can lift $\boldsymbol{y}_0$ *at precision* N *in time* $d^r\,\mathsf{R}(N)+\mathcal{O}_{N\to\infty}(N)$.

Once again, we compare with Newton iteration which performs at each step an evaluation of the polynomial equations and of their Jacobian matrix, and an inversion of its evaluated Jacobian matrix. This amounts to a cost $C\,(r\,d^r+r^\omega)\,\mathsf{I}(N)+\mathcal{O}_{N\to\infty}(N)$ where C is a constant in $\mathbb{R}_{>0}$. The latter theorem shows that we manage to save the cost of the inversion of the Jacobian matrix at full precision with on-line algorithms.

This latter advantage is meaningful when the cost of evaluation of the system is lower than the cost of linear algebra. Therefore we adapt our on-line approach to polynomials given as straight-line programs (s.l.p.), that is as a sequence of arithmetic operations without branching.

Theorem. *Let* $\boldsymbol{P}$ *be a polynomial system of* r *polynomials in* r *variables over* R, *given as an s.l.p. with* L^* *multiplications. Let* $\boldsymbol{y}_0\in(R/(p))^r$ *be such that* $\boldsymbol{P}(\boldsymbol{y}_0)=0 \bmod p$ *and* $\det\left(\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)\right)\neq 0 \bmod p$.

Then, one can lift $\boldsymbol{y}_0$ *at precision* N *in time* $3\,L^*\,\mathsf{R}(N)+\mathcal{O}_{N\to\infty}(N)$.

In this case, Newton iteration costs $C\,(L^*+r^\omega)\,\mathsf{I}(N)+\mathcal{O}_{N\to\infty}(N)$, where C is a constant in $\mathbb{R}_{\geqslant 0}$. Hence our on-line approach is particularly well-suited to systems that can be evaluated cheaply, e.g. sparse polynomial systems. Note that despite all these advantages, our algorithms are worse by a logarithmic factor in the precision N compared to zealous Newton iteration with the current implementation of the relaxed multiplication.

Finally, we implement these algorithms to obtain timings competitive with Newton and even lower on wide ranges of input parameters. Our implementation is available in the C++ library ALGEBRAMIX of MATHEMAGIX [HLM+02].

B.6 Relaxed lifting of triangular sets

We noticed in Section B.5 that relaxed algorithms could reduce the cost due to linear algebra when lifting a regular root of a polynomial system compared to previous off-line algorithms. In the same way that the Newton-Hensel operator was adapted to lift univariate representations in [GLS01, HMW01] and then triangular sets in [Sch02], we adapt our relaxed approach of Chapter 5 to lift such objects with the goal of getting rid of the contribution of linear algebra in the complexity.

Let us introduce the notion of univariate representation of a zero-dimensional ideal $\mathcal{I}\subseteq R[X_1,\ldots,X_n]$. Let A be the quotient algebra $R[X_1,\ldots,X_n]/\mathcal{I}$ and $\Lambda\in A$ such that the R-algebra $R[\Lambda]$ spanned by Λ is equal to A itself. A *univariate representation* of A consists of polynomials $\mathfrak{P}=(Q,S_1,\ldots,S_n)$ in $R[T]$ with $\deg(S_i)<\deg(Q)$ such that we have a R-algebra isomorphism

$$\begin{array}{ccc} A=R[X_1,\ldots,X_n]/\mathcal{I} & \to & R[T]/(Q) \\ X_1,\ldots,X_n & \mapsto & S_1,\ldots,S_n \\ \Lambda & \mapsto & T. \end{array}$$

The oldest trace of this representation is to be found in [Kro82] and a few years later in [Kön03]. A good summary of their work can be found in [Mac16]. The shape lemma [GM89] states the existence of such a representation for a generic linear form Λ of a zero-dimensional ideal. Different algorithms compute this representation, from a geometric resolution [GHMP97, GHH+97, GLS01, HMW01] or using a Gröbner basis [Rou99].

A *triangular set* is a set of n polynomials $\boldsymbol{t} = (t_1, ..., t_n) \subseteq R[X_1, ..., X_n]$ such that t_i is in $R[X_1, ..., X_i]$, monic in X_i and reduced with respect to $(t_1, ..., t_{i-1})$. The notion of triangular set comes from [Rit66] in the context of differential algebra. Many similar notions were introduced afterwards [Wu84, Laz91, Kal93, ALMM99]. Although all these notions do not coincide in general, they are the same for zero-dimensional ideals.

As it turns out, univariate representations can be seen as a special case of triangular sets. Indeed, with the notations above, the family $(Q(T), X_1 - S_1(T), ..., X_n - S_n(T))$ is a triangular set in the algebra $R[T, X_1, ..., X_n]$. From now on, we will consider univariate representations as a special case of triangular sets.

We define $\mathsf{Rem}(d_1, ..., d_n)$ to be the cost of arithmetic operations in the quotient algebra $R[X_1, ..., X_n]/(t_1, ..., t_n)$ with $d_i := \deg_{X_i}(t_i)$. When using univariate representations, the elements of $A \simeq R[T]/(Q)$ are represented as univariate polynomials of degree less than $d := \deg(Q)$. Then, multiplication in A costs a few polynomial multiplications.

Lifting triangular sets (or univariate representations) is a crucial operation. Several implementations of algorithms that compute triangular sets on the rationals compute these objects modulo a prime number, and then lift the representation. For example, the KRONECKER software [L+02], for univariate representations, and the REGULARCHAINS package [LMX05] of MAPLE, for triangular sets, use a lifting. Even better, another lifting is at the core of the geometric resolution algorithm [GLS01, HMW01] of KRONECKER. This algorithm manipulates many univariate representations of curves, and so requires many lifting on power series.

As it turns out, most of the time required to compute triangular sets (or univariate representations) is spent in the lifting. Therefore, any improvement on the lifting complexity will have repercussions on the whole algorithm.

Let $\boldsymbol{f} = (f_1, ..., f_n) \in R[X_1, ..., X_n]$ be a polynomial system given by an s.l.p. with L operations. If L_{f_i} is the evaluation complexity of only the output f_i, then we denote by $L^{\perp} := L_{f_1} + \cdots + L_{f_n}$ the complexity that corresponds to computing $f_1, ..., f_n$ without sharing any operations. Since $L_{f_i} \leqslant L$, we always have

$$L \leqslant L^{\perp} \leqslant n\, L.$$

An algorithm from Baur and Strassen [BS83] gives an s.l.p. that evaluates the Jacobian matrix of $\boldsymbol{f}$ in $5\, L^{\perp}$ operations.

Let $\boldsymbol{t}_0$ be a triangular set in $R/(p)[X_1, ..., X_n]$ such that $\boldsymbol{f} = 0$ in $R/(p)[X_1, ..., X_n]/\langle \boldsymbol{t}_0 \rangle$. We work under the assumption that the determinant of the Jacobian matrix $\mathrm{Jac}_{\boldsymbol{f}}$ in $\mathcal{M}_n(R/(p)[X_1, ..., X_n])$ must be invertible modulo $\boldsymbol{t}_0$. This last condition is sufficient to have the existence and uniqueness of a triangular set $\boldsymbol{t}$ in $R_p[X_1, ..., X_n]$ which reduces to $\boldsymbol{t}_0$ modulo p and satisfies $\boldsymbol{f} = 0$ in $R_p[X_1, ..., X_n]/\langle \boldsymbol{t} \rangle$.

The lifting algorithms will compute, from the inputs $\boldsymbol{f}$ and $\boldsymbol{t}_0$, this unique triangular set $\boldsymbol{t}$ at precision N.

Our contribution is to give, for any p-adic triangular set, a shifted algorithm of which it is a fixed point. Then we can apply the recursive p-adic framework and deduce a relaxed lifting algorithm for this triangular set.

For the sake of simplicity, we give the asymptotic complexities when the p-adic precision N tends to infinity. Let us now state the complexity results.

Theorem. *Our on-line algorithm can lift the triangular set $\boldsymbol{t}$ at precision N in time*

$$C\, n\, L\, \mathsf{R}(N)\, \mathsf{Rem}(d_1, \ldots, d_n) + \mathcal{O}_{N\to\infty}(N),$$

where C is a constant in $\mathbb{R}_{\geqslant 0}$.

The previous off-line algorithm of [Sch02] lift the triangular set $\boldsymbol{t}$ at precision N in time $C\,(L^{\perp} + n^{\omega})\,\mathsf{I}(N)\,\mathsf{Rem}(d_1, \ldots, d_n) + \mathcal{O}_{N\to\infty}(N)$ where C is a positive constant. Therefore, we can expect the on-line algorithm to improve the complexity of the previous algorithm only in the case $n\, L \leqslant n^{\omega}$, that is for systems $\boldsymbol{f}$ that evaluates in roughly less than n^2 operations.

The situation is more at our advantage when lifting univariate representations. We suppose in this paragraph that $\boldsymbol{t}$ is a univariate representation of degree d.

Theorem. *Our on-line algorithm can lift a univariate representation $\boldsymbol{t}$ at precision N in time*

$$C\, L\, \mathsf{R}(N)\, \mathsf{M}(d) + \mathcal{O}_{N\to\infty}(N),$$

where C is a constant in $\mathbb{R}_{\geqslant 0}$.

We compare our algorithm with the previous off-line algorithm of [GLS01, HMW01]. The off-line algorithm lifts the univariate representation $\boldsymbol{t}$ at precision N in time $C\,(L^{\perp} + n^{\omega})\,\mathsf{I}(N)\,\mathsf{M}(d) + \mathcal{O}_{N\to\infty}(N)$, where C is a positive constant. Consequently, our algorithm always does asymptotically less on-line multiplication than the other algorithm does off-line multiplications. Moreover, for systems of polynomial equations $\boldsymbol{f}$ that can be evaluated in less than n^{ω} operations, we can expect a considerable gap in performance from our algorithm.

Finally we implement these algorithms in the C++ library Algebramix of Mathemagix [HLM+02] for the special case of univariate representations. Our new relaxed algorithm compares favorably on the examples. We mention that our on-line algorithm is currently connected to Kronecker inside Mathemagix with the help of G. Lecerf.

B.7 Algorithms for the universal decomposition algebra

Let $\Bbbk$ be a field and let f in $\Bbbk[X]$ be a degree n separable polynomial. We let $\mathcal{R} := \{\alpha_1, \ldots, \alpha_n\}$ be the set of roots of f in an algebraic closure of $\Bbbk$. The *ideal of symmetric relations* $\mathcal{I}_s$ is the ideal

$$\{P \in \Bbbk[X_1, \ldots, X_n] \,|\, \forall \sigma \in \mathfrak{S}_n, P(\alpha_{\sigma(1)}, \ldots, \alpha_{\sigma(n)}) = 0\}.$$

The *universal decomposition algebra* is the quotient algebra $\mathbb{A} := \Bbbk[X_1, ..., X_n]/\mathcal{I}_s$, of dimension $\delta := n!$.

We show how to obtain efficient algorithms to compute in $\mathbb{A}$. We use a univariate representation of $\mathbb{A}$, *i.e.* an isomorphism of the form $\mathbb{A} \simeq \Bbbk[T]/Q(T)$, since in this representation, arithmetic operations in $\mathbb{A}$ are known to be quasi-optimal. We give details for two related algorithms, to find the isomorphism above, and to compute the characteristic polynomial of any element of $\mathbb{A}$, that are the first quasi-optimal algorithms for these tasks.

We measure the cost of our algorithms by the number of arithmetic operations in $\Bbbk$ they perform. Practically, this is well adapted to cases where $\Bbbk$ is a finite field; over $\Bbbk = \mathbb{Q}$, we should use lifting algorithms from Chapter 6.

The heart of the article, and the key to obtain better algorithms, is the question of which representation should be used for $\mathbb{A}$. A commonly used representation is *triangular*. The *divided differences*, also known as *Cauchy modules* [Che50, RV99], are defined by $C_1(X_1) := f(X_1)$ and

$$C_{i+1} := \frac{C_i(X_1, ..., X_i) - C_i(X_1, ..., X_{i-1}, X_{i+1})}{X_i - X_{i+1}}$$

for $1 \leqslant i < n$. They form a *triangular basis* of $\mathcal{I}_s$. Divided differences are inexpensive to compute via their recursive formula, but it is difficult to make computations in $\mathbb{A}$ efficient with this representation. The paper [BCHS11] gives a multiplication algorithm of cost $\tilde{\mathcal{O}}(\delta)$, but this algorithm hides high degree logarithmic terms in the big-O. There is no known quasi-linear algorithm for inverting elements of $\mathbb{A}$.

The second representation we discuss is univariate. When using univariate representations, the elements of $\mathbb{A} \simeq \Bbbk[T]/(Q)$ are represented as univariate polynomials of degree less than δ. Then, multiplications and inversions (when possible) in $\mathbb{A}$ cost respectively $\mathcal{O}(\mathsf{M}(\delta))$ and $\mathcal{O}(\mathsf{M}(\delta)\log(\delta))$. For characteristic polynomial, the situation is not as good, as no quasi-linear algorithm is known: the best known result [Sho94] is $\mathcal{O}(\mathsf{M}(\delta)\,\delta^{1/2} + \delta^{(\omega+1)/2})$, resulting in a $\mathcal{O}(\delta^{1.69})$ characteristic polynomial algorithm.

Computing a univariate representation for $\mathbb{A}$ is expensive: the best known algorithm [PS11] takes as input a triangular set (such as the divided differences) and convert it to a univariate representation in time $\tilde{\mathcal{O}}(\delta^{1.69})$.

Thus, the triangular representation for $\mathbb{A}$ is easy to compute but leads to rather inefficient algorithms to compute in $\mathbb{A}$. On the other hand, computing a univariate representation is not straightforward, but once it is known, some computations in $\mathbb{A}$ become faster. Our main contribution in this paper is to show how to circumvent the downsides of univariate representations, by providing fast algorithms for their construction. We also show how to use fast univariate arithmetics in $\mathbb{A}$ to compute characteristic polynomials efficiently.

For univariate representations, our algorithms are Las Vegas: we give *expected* running times.

Theorem. *Suppose that the characteristic of* $\Bbbk$ *is zero, or at least* $2\,\delta^2$*. Then we can compute characteristic polynomials and univariate representations in* $\mathbb{A}$ *with costs as specified in the following table.*

$\mathcal{X}_{P,\mathbb{A}}$	*univariate representation (expected time)*
$\mathcal{O}(n^{(\omega+3)/2}\mathsf{M}(\delta)) = \tilde{\mathcal{O}}(\delta)$	$\mathcal{O}(n^{(\omega+3)/2}\mathsf{M}(\delta)) = \tilde{\mathcal{O}}(\delta)$

We propose two approaches; both of them rely on classical ideas. The first one computes characteristic polynomials by means of their Newton sums, following previous work of [Val89, AV94, CM94], but is limited to simple polynomials, such as linear forms; this will provide the best algorithms in practice. The second one relies on iterated resultants [Lag70, Soi81, Leh97, RV99] and provides the complexity statements of the theorem.

Finally, we implement our algorithms in Magma 2.17.1 [BCP97] and give experimental results. We show practical improvements for the computation of univariate representation of $\mathbb{A}$. Our change of basis algorithms between the univariate and the triangular representation are efficient; for an operation such as inversion, even with the overhead of change of representation, it pays off to convert to a univariate representation.

B.8 Lifting of fundamental invariants

This short appendix is dedicated to prove a useful result in invariant theory, that we obtained while writing Chapter 7: it shows that so-called fundamental invariants of finite group actions always specialize well modulo all primes, except a few exceptions known in advance. This is a rare phenomenon in computer algebra, since as a rule of thumb, for non-linear systems, the primes of "bad reduction" cannot be determined in any straightforward manner.

This result has practical implications. For example, in order to compute rational primary invariants, it is sufficient to compute primary invariants modulo p. Then the lifting of primary invariants to rational coefficients is trivial.

Private communications with H. E. A. Campbell, D. Wehlau and M. Roth revealed that this result was known to them, but as far as we know, it has not appeared in print before. We chose to include it in this thesis, since it could find practical applications — to the best of our knowledge, software such as Magma [BCP97] do not make use of this kind of result in their algorithms for computing invariant rings.

This is a joint work with É. Schost.

Bibliographie

[ALMM99] P. Aubry, D. Lazard, and M. Moreno Maza. On the theories of triangular sets. *J. Symbolic Comput.*, 28(1-2):105–124, 1999. Polynomial elimination—algorithms and applications.

[AM69] M. F. Atiyah and I. G. Macdonald. *Introduction to commutative algebra*. Addison-Wesley Publishing Co., Reading, Mass.-London-Don Mills, Ont., 1969.

[ASU75] A. V. Aho, K. Steiglitz, and J. D. Ullman. Evaluating polynomials at fixed sets of points. *SIAM J. Comput.*, 4(4):533–539, 1975.

[AV94] J.-M. Arnaudiès and A. Valibouze. Calculs de résolvantes. *Rapport LITP 94.46*, 1994.

[AV97] J.-M. Arnaudiès and A. Valibouze. Lagrange resolvents. *J. Pure Appl. Algebra*, 117/118:23–40, 1997.

[AV00] P. Aubry and A. Valibouze. Using Galois ideals for computing relative resolvents. *J. Symb. Comp.*, 30(6):635–651, 2000.

[AV12] P. Aubry and A. Valibouze. Algebraic computation of resolvents without extraneous powers. *European Journal of Combinatorics*, 2012. to appear.

[BA80] R. R. Bitmead and B. D. O. Anderson. Asymptotically fast solution of Toeplitz and related systems of linear equations. *Linear Algebra Appl.*, 34:103–116, 1980.

[Bal00] W. Balser. *Formal power series and linear systems of meromorphic ordinary differential equations*. Universitext. Springer-Verlag, New York, 2000.

[BBP10] M. A. Barkatou, G. Broughton, and E. Pflügel. A monomial-by-monomial method for computing regular solutions of systems of pseudo-linear equations. *Math. Comput. Sci.*, 4(2-3):267–288, 2010.

[BCHS11] A. Bostan, M. F. I. Chowdhury, J. van der Hoeven, and É. Schost. Homotopy methods for multiplication modulo triangular sets. *J. Symb. Comp.*, 2011. To appear.

[BCL+12] A. Bostan, M. F. I. Chowdhury, R. Lebreton, B. Salvy, and É Schost. Power series solutions of singular (q)-differential equations. In *Proceedings of ISSAC'12*, pages 107–114. ACM Press, 2012.

[BCO+07] A. Bostan, F. Chyzak, F. Ollivier, B. Salvy, É. Schost, and A. Sedoglavic. Fast computation of power series solutions of systems of differential equations. In *18th ACM-SIAM Symposium on Discrete Algorithms*, pages 1012–1021, 2007. New Orleans, January 2007.

[BCP97] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).

[BCS97] P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic complexity theory*, volume 315 of *Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences]*. Springer-Verlag, Berlin, 1997. With the collaboration of Thomas Lickteig.

[Ber84] S. J. Berkowitz. On computing the determinant in small parallel time using a small number of processors. *Inform. Process. Lett.*, 18(3):147–150, 1984.

[Ber98] D. J. Bernstein. Composing power series over a finite ring in essentially linear time. *J. Symbolic Comput.*, 26(3):339–341, 1998.

[BFSS06] A. Bostan, P. Flajolet, B. Salvy, and É. Schost. Fast computation of special resultants. *J. Symb. Comp.*, 41(1):1–29, 2006.

[BGVPS05] A. Bostan, L. González-Vega, H. Perdry, and É. Schost. From Newton sums to coefficients: complexity issues in characteristic p. In *MEGA'05*, 2005.

[BHL11] J. Berthomieu, J. van der Hoeven, and G. Lecerf. Relaxed algorithms for p-adic numbers. *J. Théor. Nombres Bordeaux*, 23(3):541–577, 2011.

[BK78] R. P. Brent and H. T. Kung. Fast algorithms for manipulating formal power series. *Journal of the ACM*, 25(4):581–595, 1978.

[BL12] J. Berthomieu and R. Lebreton. Relaxed p-adic Hensel lifting for algebraic systems. In *Proceedings of ISSAC'12*, pages 59–66. ACM Press, 2012.

[BLMM01] F. Boulier, F. Lemaire, and M. Moreno Maza. Pardi! In *ISSAC'01*, pages 38–47. ACM, 2001.

[BLS03] A. Bostan, G. Lecerf, and É. Schost. Tellegen's principle into practice. In *Proceedings of ISSAC'03*, pages 37–44. ACM Press, 2003.

[BMSS08] A. Bostan, F. Morain, B. Salvy, and É. Schost. Fast algorithms for computing isogenies between elliptic curves. *Math. Comp.*, 77(263):1755–1778, 2008.

[Bos03] A. Bostan. *Algorithmique efficace pour des opérations de base en Calcul formel*. PhD thesis, École Polytechnique, 2003.

[Bou73] N. Bourbaki. *Éléments de mathématique, Fasc. XXIII*. Hermann, Paris, 1973. Livre II: Algèbre. Chapitre 8: Modules et anneaux semi-simples.

[BP99] M. Barkatou and E. Pflügel. An algorithm computing the regular formal solutions of a system of linear differential equations. *J. Symbolic Comput.*, 28(4-5):569–587, 1999. Differential algebra and differential equations.

[BS83] W. Baur and V. Strassen. The complexity of partial derivatives. *Theoret. Comput. Sci.*, 22(3):317–330, 1983.

[BS05] A. Bostan and É. Schost. Polynomial evaluation and interpolation on special sets of points. *J. Complexity*, 21(4):420–446, 2005.

[BS09] A. Bostan and É. Schost. Fast algorithms for differential equations in positive characteristic. In *ISSAC 2009—Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*, pages 47–54. ACM, New York, 2009.

[BSS08] A. Bostan, B. Salvy, and É. Schost. Power series composition and change of basis. In *ISSAC 2008*, pages 269–276. ACM, New York, 2008.

[BT80] R. P. Brent and J. F. Traub. On the complexity of composition and generalized composition of power series. *SIAM J. Comput.*, 9(1):54–66, 1980.

[CG10] A. Colin and M. Giusti. Efficient computation of squarefree lagrange resolvents. 2010.

[Che50] N. Chebotarev. *Grundzüge des Galois'shen Theorie*. P. Noordhoff, 1950.

[CK91] D. G. Cantor and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Inform.*, 28(7):693–701, 1991.

[CM94] D. Casperson and J. McKay. Symmetric functions, m-sets, and Galois groups. *Math. Comp.*, 63(208):749–757, 1994.

[Coo66] S. A. Cook. *On the minimum computation time of functions*. PhD thesis, Harvard University, 1966.

[CS04] Z. Chen and A. Storjohann. IML, the Integer Matrix Library, 2004. Version 1.0.3.

[CT65] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comp.*, 9(3):251–280, 1990.

[DFS09] L. De Feo and É. Schost. Fast arithmetics in Artin-Schreier towers over finite fields. In *ISSAC'09*, pages 127–134. ACM, 2009.

[DFS10] L. De Feo and É. Schost. transalpyne: a language for automatic transposition. *ACM Commun. Comput. Algebra*, 44(1/2):59–71, July 2010.

[DIS11] C.-É. Drevet, M. N. Islam, and É. Schost. Optimization techniques for small matrix multiplication. *Theoret. Comput. Sci.*, 412(22):2219–2236, 2011.

[Dix82] J. D. Dixon. Exact solution of linear equations using p-adic expansions. *Numer. Math.*, 40(1):137–141, 1982.

[DK02] H. Derksen and G. Kemper. *Computational invariant theory*. Invariant Theory and Algebraic Transformation Groups, I. Springer-Verlag, Berlin, 2002. Encyclopaedia of Mathematical Sciences, 130.

[DKSS08] A. De, P. Kurur, C. Saha, and R. Saptharishi. Fast integer multiplication using modular arithmetic. In *STOC'08*, pages 499–505. ACM, New York, 2008.

[DL08] C. Durvye and G. Lecerf. A concise proof of the Kronecker polynomial system solver from scratch. *Expo. Math.*, 26(2):101–139, 2008.

[DMMSX06] X. Dahan, M. Moreno Maza, É. Schost, and Y. Xie. On the complexity of the D5 principle. In *Transgressive Computing*, pages 149–168, 2006.

[FGLM93] J. C. Faugère, P. Gianni, D. Lazard, and T. Mora. Efficient computation of zero-dimensional Gröbner bases by change of ordering. *J. Symb. Comp.*, 16(4):329–344, 1993.

[FM11] J.-C. Faugère and C. Mou. Fast algorithm for change of ordering of zero-dimensional Gröbner bases with sparse multiplication matrices. In *ISSAC'11*, pages 115–122. ACM, 2011.

[FS74] M. J. Fischer and L. J. Stockmeyer. Fast on-line integer multiplication. *J. Comput. System Sci.*, 9:317–331, 1974.

[Für07] M. Fürer. Faster Integer Multiplication. In *Proceedings of STOC 2007*, pages 57–66, San Diego, California, 2007.

[G+91] T. Granlund et al. GMP, the GNU multiple precision arithmetic library, 1991. Version 5.0.2.

[GG03] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, Cambridge, second edition, 2003.

[GHH+97] M. Giusti, J. Heintz, K. Hägele, J. E. Morais, L. M. Pardo, and J. L. Montaña. Lower bounds for Diophantine approximations. *J. Pure Appl. Algebra*, 117/118:277–317, 1997. Algorithms for algebra (Eindhoven, 1996).

[GHMP97] M. Giusti, J. Heintz, J. E. Morais, and L. M. Pardo. Le rôle des structures de données dans les problèmes d'élimination. *C. R. Acad. Sci. Paris Sér. I Math.*, 325(11):1223–1228, 1997.

[GLS01] M. Giusti, G. Lecerf, and B. Salvy. A Gröbner free alternative for polynomial system solving. *J. Complexity*, 17(1):154–211, 2001.

[GM89] P. Gianni and T. Mora. Algebraic solution of systems of polynomial equations using Groebner bases. In *Applied algebra, algebraic algorithms and error-correcting codes (Menorca, 1987)*, volume 356 of *Lecture Notes in Comput. Sci.*, pages 247–257. Springer, Berlin, 1989.

[GR08] V. Guruswami and A. Rudra. Explicit codes achieving list decoding capacity: error-correction with optimal redundancy. *IEEE Trans. Inform. Theory*, 54(1):135–150, 2008.

[Har12] D. Harvey. The Karatsuba middle product for integers. to appear, 2012.

[Hen18] K. Hensel. Eine neue Theorie der algebraischen Zahlen. *Math. Z.*, 2(3-4):433–452, 1918.

[Hen66] F. C. Hennie. On-line turing machine computations. *Electronic Computers, IEEE Transactions on*, EC-15(1):35 –44, 1966.

[HKP+00] J. Heintz, T. Krick, S. Puddu, J. Sabia, and A. Waissbein. Deformation techniques for efficient polynomial equation solving. *J. Complexity*, 16(1):70–109, 2000.

[HLM+02] J. van der Hoeven, G. Lecerf, B. Mourrain, et al. Mathemagix, 2002. SVN Version 7058. Available from `http://www.mathemagix.org`.

[HMW01] J. Heintz, G. Matera, and A. Waissbein. On the time-space complexity of geometric elimination procedures. *Appl. Algebra Engrg. Comm. Comput.*, 11(4):239–296, 2001.

[Hoe97] J. van der Hoeven. Lazy multiplication of formal power series. In *ISSAC '97*, pages 17–20, Maui, Hawaii, 1997.

[Hoe02] J. van der Hoeven. Relax, but don't be too lazy. *J. Symb. Comput.*, 34(6):479–542, 2002.

[Hoe03] J. van der Hoeven. Relaxed multiplication using the middle product. In *Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation*, pages 143–147 (electronic), New York, 2003. ACM.

[Hoe07] J. van der Hoeven. New algorithms for relaxed multiplication. *J. Symbolic Comput.*, 42(8):792–802, 2007.

[Hoe09] J. van der Hoeven. Relaxed resolution of implicit equations. Technical report, HAL, 2009.

[Hoe11] J van der Hoeven. From implicit to recursive equations. Technical report, HAL, 2011.

[Hoe12] J. van der Hoeven. Faster relaxed multiplication. Technical report, HAL, 2012.

[HQZ04] G. Hanrot, M. Quercia, and P. Zimmermann. The middle product algorithm. I. *Appl. Algebra Engrg. Comm. Comput.*, 14(6):415–438, 2004.

[HZ04] G. Hanrot and P. Zimmermann. A long note on Mulders' short product. *J. Symbolic Comput.*, 37(3):391–401, 2004.

[Kal92] E. Kaltofen. On computing determinants of matrices without divisions. In P. S. Wang, editor, *Proc. 1992 Internat. Symp. Symbolic Algebraic Comput. (ISSAC'92)*, pages 342–349, New York, N. Y., 1992. ACM Press.

[Kal93] M. Kalkbrener. A generalized Euclidean algorithm for computing triangular representations of algebraic varieties. *J. Symb. Comp.*, 15:143–167, 1993.

[Kap01] G. Kapoulas. Polynomially time computable functions over p-adic fields. In *Computability and complexity in analysis*, volume 2064, pages 101–118. Springer, Berlin, 2001.

[Kar97] J. Karczmarczuk. Generating power of lazy semantics. *Theoret. Comput. Sci.*, 187(1-2):203–219, 1997.

[Kat90] S. Katsura. Spin glass problem by the method of integral equation of the effective field. *New Trends in Magnetism*, pages 110–121, 1990.

[Kir01] Peter Kirrinnis. Fast algorithms for the Sylvester equation $A\,X - X\,B^{\langle \mathsf{ssf} \rangle T} = C$. *Theoret. Comput. Sci.*, 259(1-2):623–638, 2001.

[KO63] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595, 1963.

[Kön03] J. König. *Aus dem Ungarischen übertragen vom Verfasser*. B. G. Teubner, Leipzig, 1903.

[Kro82] L. Kronecker. Grundzüge einer arithmetischen theorie des algebraischen grössen. *J. reine angew. Math.*, 92:1–122, 1882.

[KU11] K. S. Kedlaya and C. Umans. Fast polynomial factorization and modular composition. *SIAM J. Comput.*, 40(6):1767–1802, 2011.

[KV04] E. Kaltofen and G. Villard. On the complexity of computing determinants. *Comput. Complexity*, 13(3-4):91–130, 2004.

[L+02] G. Lecerf et al. Kronecker, 2002. Available from `http://lecerf.perso.math.cnrs.fr/software/kronecker/`.

[Lag70] J.-L. Lagrange. Réflexions sur la résolution algébrique des équations. *Mémoires de l'Académie de Berlin*, 1770.

[Lan91] L. Langemyr. Algorithms for a multiple algebraic extension. In *Effective methods in algebraic geometry)*, volume 94 of *Progr. Math.*, pages 235–248. Birkhäuser, 1991.

[Laz91] D. Lazard. A new method for solving algebraic systems of positive dimension. *Discrete Appl. Math.*, 33(1-3):147–160, 1991. Applied algebra, algebraic algorithms, and error-correcting codes (Toulouse, 1989).

[Leh97] F. Lehobey. Resolvent computations by resultants without extraneous powers. In *ISSAC '97*, pages 85–92. ACM, 1997.

[Lin08] The LinBox Group. *LinBox – Exact Linear Algebra over the Integers and Finite Rings*, 2008. SVN Version 4136.

[LMMS09] X. Li, M. Moreno Maza, and É. Schost. Fast arithmetic for triangular sets: from theory to practice. *J. Symbolic Comput.*, 44(7):891–907, 2009.

[LMP09] X. Li, M. Moreno Maza, and W. Pan. Computations modulo regular chains. In *ISSAC'09*, pages 239–246. ACM, 2009.

[LMS09] X. Li, M. Moreno Maza, and É. Schost. Fast arithmetic for triangular sets: from theory to practice. *Journal of Symbolic Computation*, 44(7):891–907, 2009.

[LMS12] R. Lebreton, E. Mehrabi, and É Schost. On the complexity of computing certain resultants. Poster at ISSAC'12, 2012.

[LMX05] F. Lemaire, M. Moreno Maza, and Y. Xie. The `RegularChains` library. In Ilias S. Kotsireas, editor, Maple Conference 2005, pages 355–368, 2005.

[LS08] R. Lercier and T. Sirvent. On Elkies subgroups of l-torsion points in elliptic curves defined over a finite field. *J. Théor. Nombres Bordeaux*, 20(3):783–797, 2008.

[LS12] R. Lebreton and É. Schost. Algorithms for the universal decomposition algebra. In *Proceedings of ISSAC'12*, pages 234–241. ACM Press, 2012.

[Mac16] F. S. Macaulay. *The algebraic theory of modular systems*. Cambridge University Press, 1916.

[MC79] R. T. Moenck and J. H. Carter. Approximate algorithms to derive exact solutions to systems of linear equations. In *EUROSAM '79*, volume 72, pages 65–73. Springer, 1979.

[Mor74] M. Morf. *Fast algorithms for multivariable systems*. PhD thesis, Stanford University, 1974.

[Mor80] M. Morf. Doubling algorithms for toeplitz and related equations. In *IEEE Conference on Acoustics, Speech, and Signal Processing*, pages 954–959, 1980.

[Mul00] T. Mulders. On short multiplications and divisions. *Appl. Algebra Engrg. Comm. Comput.*, 11(1):69–88, 2000.

[New36] I. Newton. *The method of fluxions and infinite series: with its application to the geometry of curve-lines*. Henry Woodfall, 1736.

[Pan01] V. Y. Pan. *Structured matrices and polynomials*. Birkhäuser Boston Inc., Boston, MA, 2001. Unified superfast algorithms.

[PS11] A. Poteaux and É. Schost. On the complexity of computing with zero-dimensional triangular sets. *Submitted*, 2011.

[Ren04] N. Rennert. A parallel multi-modular algorithm for computing Lagrange resolvents. *J. Symbolic Comput.*, 37(5):547–556, 2004.

[Rit66] J. F. Ritt. *Differential algebra*. Dover Publications Inc., New York, 1966.

[Rou99] F. Rouillier. Solving zero-dimensional systems through the rational univariate representation. *Appl. Algebra Engrg. Comm. Comput.*, 9(5):433–461, 1999.

[RV99] N. Rennert and A. Valibouze. Calcul de résolvantes avec les modules de Cauchy. *Exp. Math.*, 8(4):351–366, 1999.

[S+90] V. Shoup et al. NTL: a library for doing number theory, 1990. Version 5.5.2. Available from `http://www.shoup.net/ntl/`.

[Sch80] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.

[Sch82] A. Schönhage. The fundamental theorem of algebra in terms of computational complexity. Technical report, Univ. Tübingen, 1982. 73 pages.

[Sch97] M. Schröder. Fast online multiplication of real numbers. In *STACS 97 (Lübeck)*, volume 1200 of *Lecture Notes in Comput. Sci.*, pages 81–92. Springer, Berlin, 1997.

[Sch02] É. Schost. Degree bounds and lifting techniques for triangular sets. In *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, pages 238–245 (electronic), New York, 2002. ACM.

[Ser78] J.-P. Serre. *Représentations linéaires des groupes finis*. Hermann, Paris, revised edition, 1978.

[Sho94] V. Shoup. Fast construction of irreducible polynomials over finite fields. *J. Symb. Comp.*, 17(5):371–391, 1994.

[Soi81] L. Soicher. *The computation of the Galois groups*. PhD thesis, Concordia University, Montreal, Quebec, Canada, 1981.

[SS71] A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, 7:281–292, 1971.

[Sto03] A. Storjohann. High-order lifting and integrality certification. *J. Symbolic Comput.*, 36(3-4):613–648, 2003. ISSAC'2002, Lille.

[Sto05] A. Storjohann. The shifted number system for fast linear algebra on integer matrices. *J. Complexity*, 21(4):609–650, 2005.

[Sto10] A. Stothers. *On the Complexity of Matrix Multiplication*. PhD thesis, University of Edinburgh, 2010.

[Stu93] B. Sturmfels. *Algorithms in invariant theory*. Texts and Monographs in Symbolic Computation. Springer-Verlag, Vienna, 1993.

[Too63] A. L. Toom. The complexity of a scheme of functional elements simulating the multiplication of integers. *Dokl. Akad. Nauk SSSR*, 150:496–498, 1963.

[Val89] A. Valibouze. Fonctions symétriques et changements de bases. In *EUROCAL'87*, volume 378 of *LNCS*, pages 323–332, 1989.

[VW11] V. Vassilevska Williams. Breaking the Coppersmith-Winograd barrier. 2011.

[Wak70] A. Waksman. On Winograd's algorithm for inner products. *IEEE Trans. Computers*, C-19(4):360–361, 1970.

[Was65] W. Wasow. *Asymptotic expansions for ordinary differential equations*. Pure and Applied Mathematics, Vol. XIV. Interscience Publishers John Wiley & Sons, Inc., New York-London-Sydney, 1965.

[Wat89] S. Watt. A fixed point method for power series computation. In P. Gianni, editor, *Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 206–217. Springer Berlin / Heidelberg, 1989.

[Wu84] W. J. Wu. Basic principles of mechanical theorem proving in elementary geometries. *J. Systems Sci. Math. Sci.*, 4(3):207–235, 1984.

[Yok97] K. Yokoyama. A modular method for computing the Galois groups of polynomials. *J. Pure Appl. Algebra*, 117/118:617–636, 1997.

[Zip79] R. Zippel. Probabilistic algorithms for sparse polynomials. In *EUROSAM'79*, volume 72 of *LNCS*, pages 216–226. Springer, 1979.

Résumé

Contributions à l'algorithmique détendue et
à la résolution des systèmes polynomiaux

Cette thèse est en majeure partie dédiée au calcul rapide de remontée p-adique par des algorithmes détendus.

Dans une première partie, nous présentons le cadre général des algorithmes détendus et de leur application au calcul de p-adiques récursifs. Pour appliquer ce cadre à la remontée p-adique de divers systèmes d'équations, il reste à transformer ces équations implicites en équations récursives. Ainsi, la seconde partie traite des systèmes d'équations linéaires, éventuellement différentiels. La remontée de résolutions de systèmes polynomiaux se trouve en troisième partie. Dans tous les cas, les nouveaux algorithmes détendus sont comparés, en théorie comme en pratique, aux algorithmes existants.

En quatrième partie, nous étudions l'algèbre de décomposition universelle d'un polynôme. Nous développons un algorithme rapide pour calculer une représentation adéquate de cette algèbre et l'utilisons pour manipuler efficacement les éléments de l'algèbre.

Finalement, nous montrons en annexe que la recherche d'invariants fondamentaux d'algèbres d'invariants sous un groupe fini peut se faire directement modulo p, facilitant ainsi leur calcul.

Mots clés. Algorithme détendu, résolution de systèmes polynomiaux, remontée p-adique.

Abstract

Contributions to relaxed algorithms and polynomial system solving

This PhD thesis is mostly devoted to the computation of p-adic lifting by relaxed algorithms.

In a first part, we introduce relaxed algorithms and their application to the computation of recursive p-adics. In order to use this framework for the p-adic lifting of various systems of equations, we have to transform the given implicit equations into recursive equations. The case of systems of linear equations, possibly differential, is treated in the second part. This third part contains the lifting of resolutions of polynomial systems. In any cases, these new relaxed algorithms are compared, both in theory and practice, to existing algorithms.

In the fourth part, we focus on the universal decomposition algebra. We present a fast algorithm which computes an adequate representation of this algebra and use it to compute efficiently with the elements of this algebra.

Finally, we show in the appendix that finding fundamental invariants of polynomial invariants algebras under a finite group can be done directly modulo p, hence making their computation easier.

Keywords. Relaxed algorithm, polynomial system solving, p-adic lifting.